NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A MODEL FOR MERGING DIFFERENT VERSIONS OF A PSDL PROGRAM

by

David Anthony Dampier

Thesis Advisor: June 1990 Valdis Berzins

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School 6b. OFFICE SYMBOL (if applicable) 52	7a. NAME OF MONITORING ORGANIZATION National Science Foundation
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code) Washington, D.C. 20550
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER O&MN, Direct funding
8c. ADDRESS (City, State, and ZIP Code) Monterey CA 93943	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A MODEL FOR MERGING DIFFERENT VERSIONS OF A PSDL PROGRAM

12. PERSONAL AUTHOR(S)
DAMPIER, DAVID ANTHONY

13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM 10/89 TO 06/90	14. DATE OF REPORT (Year, Month, Day) June 1990	15 PAGE COUNT 101

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
rapid prototyping, models, programming languages, merging, automatic change propagation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

As hardware complexity increases, software complexity increases, and software systems become less maintainable by manual methods. Automated software development methods, like Rapid Prototyping, have served to increase the maintainability of modern software systems, and increase customer participation in the requirements definition process. This makes software systems more maintainable and increases customer satisfaction with the first version of the system. Still, changes are inevitable. The part of the maintenance problem that automated tools currently do not address, is the automatic propagation of changes through multiple versions of the same system.

The Prototype System Description Language (PSDL) is a language used exclusively for designing and executing rapid prototypes. This thesis is directed at developing a model for automatically merging two different versions of a PSDL program, providing a method for propagating changes through multiple versions of that program.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Berzins, Valdis	22b. TELEPHONE (Include Area Code) (408) 646-2735	22c. OFFICE SYMBOL 52Be

83 APR edition may be used until exhausted
All other editions are obsolete

Approved for public release; distribution is unlimited

A MODEL FOR
MERGING DIFFERENT VERSIONS OF
A PSDL PROGRAM

by
David Anthony Dampier
Captain, United States Army
B.S., University of Texas at El Paso, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

Author:
David Anthony Dampier

Approved By: Valdis Berzins
Valdis Berzins, Thesis Advisor

Luqi
Luqi, Second Reader

Robert B. McGhee
Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

As hardware complexity increases, software complexity increases, and software systems become less maintainable by manual methods. Automated software development methods, like Rapid Prototyping, have served to increase the maintainability of modern software systems, and increase customer participation in the requirements definition process. This makes software systems more maintainable and increases customer satisfaction with the first version of the system. Still, changes are inevitable. The part of the maintenance problem that automated tools currently do not address, is the automatic propagation of changes through multiple versions of the same system.

The Prototype System Description Language (PSDL) is a language used exclusively for designing and executing rapid prototypes. This thesis is directed at developing a model for automatically merging two different versions of a PSDL program, providing a method for propagating changes through multiple versions of that program.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS

$\equiv$	--	Is Equivalent To
$\in$	--	Is an Element Of
$\neg$	--	Logical Negation
$\Rightarrow$	--	Logical Implication
$\Leftrightarrow$	--	If and Only If/Logical Equivalence
$\wedge$	--	Logical And
$\vee$	--	Logical Or
$\top$	--	Top/Inconsistent
$\bot$	--	Bottom/Undefined
$\exists$	--	There Exists At Least One
$\forall$	--	For All Elements
$\cap$	--	Set Intersection
$\cup$	--	Set Union
$-$	--	Set Difference
$\sqcap$	--	Greatest Common Approximation
$\sqcup$	--	Least Common Extension
$\sqsubseteq$	--	Approximates

ACKNOWLEDGEMENT

I would like to thank my wife Caryn for her unselfish support during these last twenty-two months. She has been very loving and understanding, and without her support, I would never have maintained my sanity. I would also like to thank Dr. Berzins and Dr. Luqi for their patience and perserverance. Without their guidance, I would not understand half of what I have done.

During the course of doing this thesis, I have learned a tremendous amount about applied mathematics and abstract algebra that I never knew existed. I have come much farther than I ever expected, and now, only a fraction of the distance I want to go.

I. INTRODUCTION

Software Development is an ever-increasing and complex industry. As hardware technology gains sophistication, so must the software that drives it. In the 1960's, IBM developed OS/360. It has been reported that it took 5000 person-years to develop and document that system.[Ref. 1] Since then, hardware has become even more complex. It has been said that the software needed to operate the Strategic Defense Initiative Systems will far exceed ten million lines of code. [Ref. 2] With software systems that sophisticated, it is easy to see that current software development methods are not adequate to ensure their reliability. To meet this challenge, automated software development methods must be developed which will increase reliability far beyond what it is today.

Another factor in the need for automated software development methods is the inability of most customers to precisely state their needs in the early stages of a system's life-cycle. This emphasizes the need for getting the customer more involved in the early stages of software development, and a method for providing the customer with a more accurate feeling about what the software system will do when it is finished as early as possible. To do this, there must be an easy way for making changes to software systems throughout the system lifetime, from the earliest stages of conceptual design through system retirement. This, in turn, means that software must be developed with evolution in mind.

A. SOFTWARE MAINTENANCE

Maintenance can be defined as modification to a software product after delivery to correct faults, improve performance or other attributes, or adapt it to a changec environment. [Ref. 3] This mindset about maintenance being done only after the system is delivered is the traditional notion, and it is one of the reasons maintenance is so hard. Designers using that mentality were not forced to design their code in a way that made making changes easy. According to [Ref. 3], the average system in use in 1987 was three to four years old and consisted of approximately 55 separate programs and 23,000 different source statements. Maintenance on these systems is hard because most of them were not well documented. when they were designed, and the people who built them are no longer available to maintain them. This means that a massive effort is needed to figure out how to change some of them. A different mindset is needed from the first stages of conceptual design.

We prefer to think of software evolution as any modification made to a software product throughout its life-cycle. Consequently, software should be designed with evolution as a primary consideration. Some of the factors which can facilitate evolution are:

1. Modularity - Modules should be designed to be as self-sufficient as possible.

2. Readability - Structured programming techniques should be used whenever possible to compensate for a wide variety of programming styles.

3. Documentation - Design decisions as well as code should be well documented and the documentation should be maintained throughout the life of the system.

4. Simplicity - Designs should be made as simple as possible. The more complex a system is, the harder it becomes to understand.

These factors are not sufficient to ensure system maintainability, but they are necessary. As we mentioned earlier, software systems in the future are going to be much larger and more complex. Traditional methods for design and maintenance will not be sufficient. More innovative ways to handle these problems will be necessary.

1. Model for Software Manufacture

Software manufacture can be defined as the combining of primitive components of a software system, through a sequence of derivations, into one or more software products. It is software manufacture which establishes the relationships between the components of a software system and the primitives from which they were derived. Within the software manufacturing process, one of the most difficult problems involves dealing with change.

Changes are inevitable, and they can come in several different ways. They can be "pre-planned", as there may not have been enough time to incorporate all of the proposed capabilities into a system in the time provided, they can be "corrective", if a bug is discovered in the system, or they can be "opportunistic", when it is discovered that a change can easily be made to the original design which will improve it in some way. When changes are made, problems can develop if all the components of the system affected by the change are not modified to deal with the change. As systems get larger and more sophisticated, these problems are amplified. [Ref. 4]

In [Ref. 4], a model is presented for managing the software manufacturing process. It is designed to represent software systems at a very low level, concentrating on what the system does and how its components depend on one another. The model has two parts, the configuration, ζ and a set of difference predicates, which serve to insure consistent change incorporation.

The configuration, ζ, consists of a bipartite Directed Acyclic Graph (DAG), with components[1] in one set and manufacturing steps in the other. Manufacturing steps are non-concrete derivations and processes performed on components. ζ is a tuple $< \mathbf{G}, \mathbf{E}, \mathbf{L} >$ where $\mathbf{G}$ is a DAG, $\mathbf{E}$ is the subset of ζ which contains the export components of the system, and $\mathbf{L}$ is a labeling function which assigns distinct labels to all the nodes in $\mathbf{G}$.

[1] Components in this model are software components, development tools, etc., which exist in the manufacturing environment.

This is a fairly good model. It provides an excellent representation of historical data, which is useful for managing change information. A problem arises when two separate versions of a system have emerged and the merging of their capabilities is desired. It also does not provide for the propagation of changes throughout a series of different versions of a system. Each such update must be made individually. Additionally, the graph contains too many unnecessary components. A much simpler model which solves some of these problems is presented in the next section.

2. Model For Software Maintenance

The model for software maintenance contained in [Ref. 5] is designed to provide a methodology for integrating information about software maintenance activities and configuration control. This model assumes the following:

1. Management controls system changes.

2. Actual maintenance is performed outside the central configuration repository.

3. Products of the configuration are derived from the repository and installed at the production site.

4. The system configuration remains consistent at all times.

The model is comprised of two major elements, system components and maintenance steps. System components are defined as immutable and non-re-derivable[2] software objects. Maintenance steps are defined as activities which can change the configuration of the system.

The configuration is modelled as a bipartite DAG **G** of components (C nodes) and maintenance steps (M nodes), connected by a set of input arcs, I and a set of output arcs, O. A sample configuration is shown in Figure 1. Maintenance steps are labelled M_q, where q is the number of the step, using simple enumeration. the input and output sets of M_q are labelled I_{Mq} and O_{Mq}, respectively. The following properties apply to maintenance steps:

1. All maintenance steps can have 0 or more inputs, no more than one output. $\forall \mathbf{M}, \mid \mathbf{I_M} \mid \geq 0 \;\&\; \mid \mathbf{O_M} \mid \leq 1$

2. *A maintenance step is empty if and only if it has no* inputs and no outputs. $\mid \mathbf{I_M} \mid = \mid \mathbf{O_M} \mid = 0$

3. If a maintenance step has at least one input, it must have an output. $\forall \mathbf{M}, \mid \mathbf{I_M} \mid \neq 0 \Longrightarrow \mid \mathbf{O_M} \mid = 1$

4. A component cannot be in the input and output sets of the same maintenance step. $c \in \mathbf{O_{Mq}} \Longrightarrow \neg(c \in \mathbf{I_{Mq}})$

5. No one component can be the output of two different maintenance steps. $\forall m_i, m_j \in \mathbf{M}, (\exists c \in \mathbf{C}$ such that $((m_i, c) \in \mathbf{O} \;\&\; (m_j, c) \in \mathbf{O})) \Longrightarrow m_i = m_j$

6. There is a set of primitives in the configuration which are not outputs of any maintenance step. $\mathbf{P} = \{c \in \mathbf{C} \mid \neg\exists\, m \in \mathbf{M}$ such that $(m, c) \in \mathbf{O}\}$

[2] Non-re-derivable objects are source objects, while re-derivable objects are those objects which can be constructed by applying some tool, or set of tools, to a set of source objects.

7. Let $\mathbf{D}^* = (\mathbf{I} \cup \mathbf{O})^*$ be the reflexive transitive closure of the union of the input and output relations **I** and **O**, then:

a. One component depends on another if and only if they share an edge in $\mathbf{D}^*$ or they are both primitive components and related by an "is-component-of" dependency.

c_j depends on $c_i \Leftrightarrow (c_i, c_j) \in \mathbf{D}^* \vee$

$c_i, c_j \in \mathbf{P}$ such that is-component-of(c_j, c_i)

b. One maintenance step depends on another if and only if they share an edge in $\mathbf{D}^*$. m_j depends on $m_i \Leftrightarrow (m_i, m_j) \in \mathbf{D}^*$

c. $\mathbf{M}_c$ is the set of maintenance steps affected by a change to the component c. $\mathbf{M}_c = \{m \in \mathbf{M} \mid (c, m) \in \mathbf{D}^*$

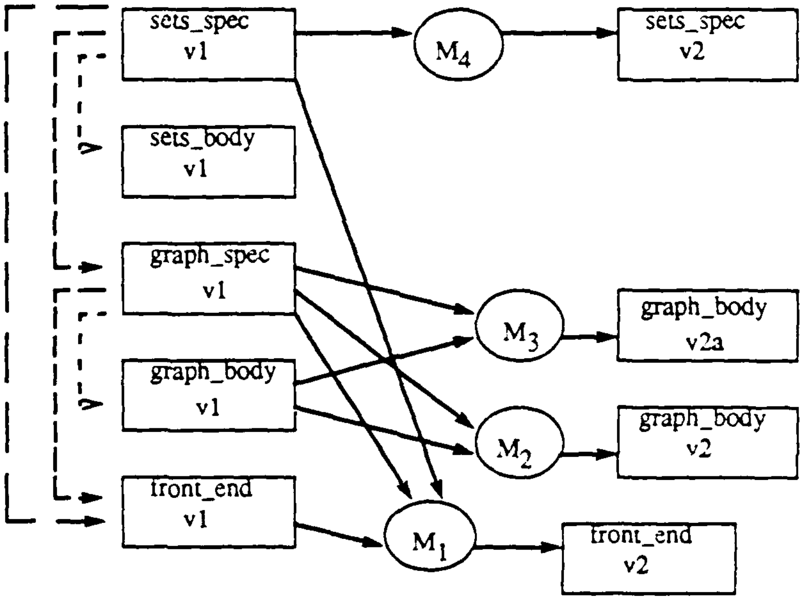

Figure 1. Sample Configuration in The Model for Software Maintenance.

Maintenance steps can be in one of five possible states:

1. *Invoked* - A requirement for the step has been identified, and is under analysis.

2. *Pending* - The step has been approved, but assets have not been obligated to perform the step.

3. *Implementing* - The work is under way.

4. *Completed* - The work is finished and the output has been returned to the central repository.

5. *Abandoned* - The work was stopped before completion or the step was never approved.[3]

If a maintenance step is in the implementing state, it can be turned back to the pending state, however all work done on this maintenance step is lost and must be redone. This is impractical, because composite maintenance steps can be composed of many smaller maintenance steps, and if a large majority of the component steps are completed when the composite step is turned back to the pending state, all of that work is lost.

An atomic maintenance step is defined as a single change in a single component, its primary input. A component c_j is a direct descendant of a component c_i if $c_j \in \mathbf{O}_{\mathbf{M}_q}$ and c_i is the primary input of $\mathbf{M}_q$, or the primary input of $\mathbf{M}_q$ is a direct descendant of c_i. This recursive direct descendance relationship defines evolution genealogy sub-graphs of $\mathbf{G}$ as

[3] A maintenance step can be abandoned at any time in the first three states. Once it is completed, it stays in the configuration forever.

trees with primary inputs and their direct descendants. The genealogy trees have the following properties:

1. All direct descendants of a component *c* belong to the genealogy tree, or sub-tree, which has *c* as its root.

2. There exists a unique path between *c* and any of its direct descendants.

When multiple maintenance steps use a component *c* as their primary input, then parallel genealogies are formed. The outputs of these parallel genealogies become different versions of a system derived from a common base, *c*. One of the problems with this model is that it provides no way for these different versions to be integrated back together to capture all of the capabilities of both genealogies in one component. This idea provides part of the motivation for this thesis.

B. RAPID PROTOTYPING

Rapid prototyping is intended to allow the user to get a better handle on exactly what his/her requirements are early in the conceptual design phase of development. It involves the use of automated tools to rapidly create "a concrete executable model of selected aspects of a proposed system"[Ref. 6] to allow the user to view the model and make comments early. The prototype is then rapidly reworked and redemonstrated to the user over several iterations until the designer and the user have a precise view of what the system should do. This process produces a validated set of requirements which become the basis for designing the final product.[Ref. 6] The prototype can also become part of the final product. In some prototyping methodologies, the prototype is an executable shell of the final system, containing only a subset of the system's ultimate functionality. After the prototype is approved by the customer, the holes are filled in and the system is delivered. In this approach to rapid prototyping, software systems can be delivered incrementally as parts of the system become fully operational.[Ref. 6] Figure 2 shows the life-cycle model for this prototyping methodology.

C. COMPUTER AIDED PROTOTYPING SYSTEM

A set of computer-aided software development tools, called the Computer-Aided Prototyping System or CAPS is being developed at the Naval Postgraduate School to support prototyping of embedded hard real-time systems.[Ref. 6] CAPS is designed to reduce the amount of effort required by the prototype designer, by providing an integrated set of tools, snown in Figure 3, to help design, translate and execute the prototypes, along with a language in which to design and program the prototypes.

Computer-aided software development tools are what puts the word "rapid" in rapid prototyping. The tools provided for in CAPS are divided into three categories:

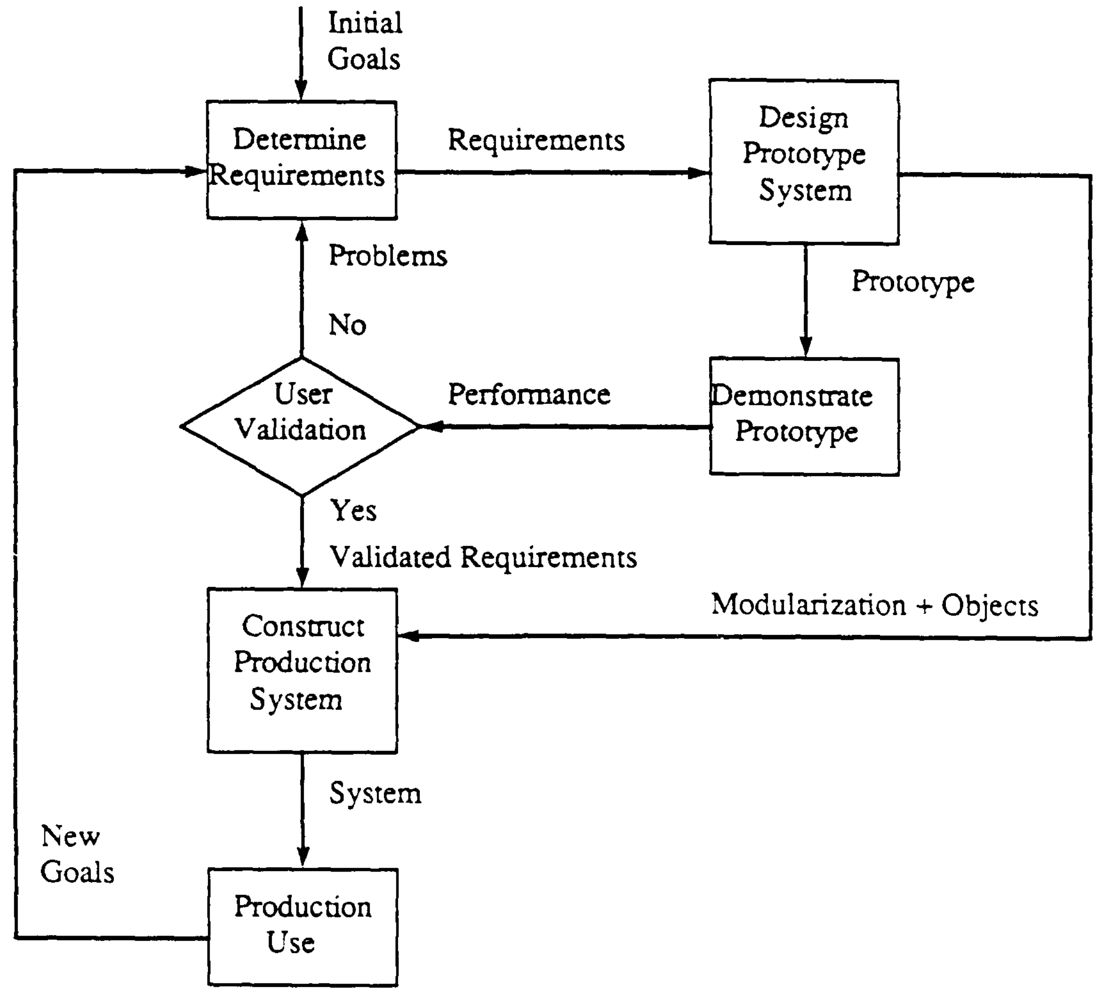

Figure 2. The Prototyping Life-Cycle Model.

1. User Interface
2. Software Database System
3. Execution Support System

The user interface contains tools that support the prototype designer in designing and programming prototypes.

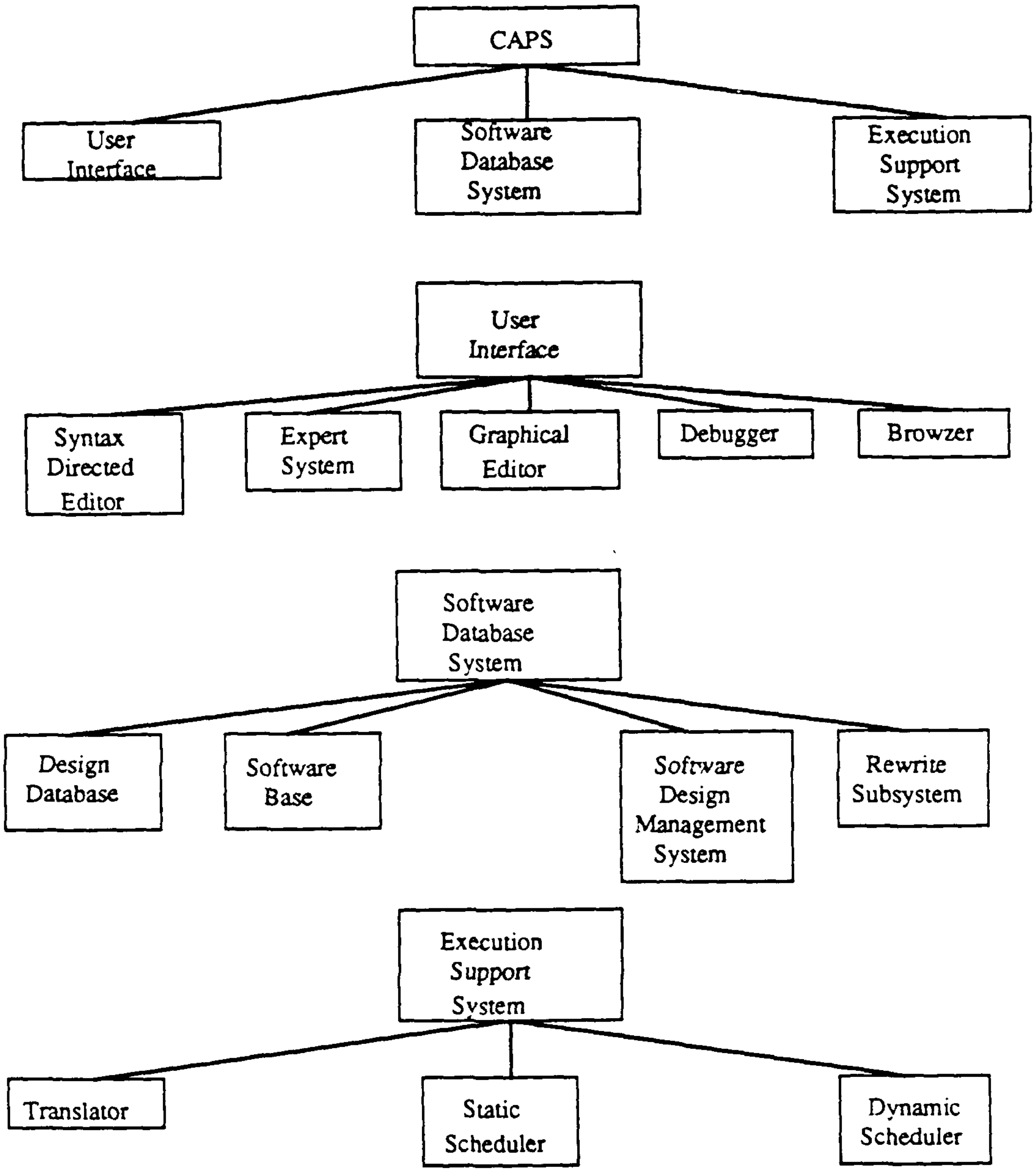

Figure 3. Computer-Aided Prototyping System Tools. [Ref. 8]

the software database system provides tools which search the software base for reusable components, retrieve them, and make them ready for use by the designer in a prototype. The execution support system provides tools which translate, schedule, and execute the prototype for the designer. The prototypes are written in a language designed specifically for CAPS, called PSDL.

D. PROTOTYPE SYSTEM DESCRIPTION LANGUAGE

PSDL [Ref. 7] is the design based language written specifically for CAPS, to provide the designer with a simple way to abstractly specify software systems. PSDL places strong emphasis on modularity, simplicity, reuse, adaptability, abstraction, and requirements tracing. [Ref. 8]

Modularity is supported through the use of independent operators which can only gain access to other operators when they are connected via data streams. Operators can represent either functions or state machines, depending on whether or not they have state variables. Data streams can be one of two types, data flow streams or sampled streams. Data flow streams operate like FIFO queues of size one. Once a value is placed on the stream, it must be read before another value can be placed on the stream. Sampled streams operate like memory cells of size one. A value is on the stream until it is replaced by another value. It is possible, with sampled streams, that some values could be read more than once, and some may never be read, because they are replaced before the stream is sampled.

Simplicity is gained through the use of a small number of language constructs which provide powerful capabilities for designing and retrieving prototypes. The grammar for the current implementation of the language, located in Appendix A, was taken from [Ref. 9] and updated with changes made since the publication of that source.

PSDL prototypes are adaptable through the use of control constraints. constraints can be placed on the inputs and outputs of operators, as well as timing requirements. Reusable software components can be modified slightly, when retrieved, to conform to these constraints.[Ref. 8]

Requirements can be traced in prototypes through the use of description constructs which can be written to reflect the requirements used in their design.

PSDL programs are written as a set of PSDL operators and data types, containing zero or more of each. PSDL operators consist of a specification and an implementation. The specification defines the external interfaces of the operator through a series of interface declarations, provides timing constraints, and describes the functionality of the operator through the use of formal and informal descriptions. The implementation can either be in PSDL or Ada. Ada implementations are Ada software objects which provide the functionality required by the operator specification. PSDL implementations are data flow diagrams augmented with a set of data stream definitions and a set of control constraints. PSDL types also contain a specification and an implementation.

E. CHANGE PROPAGATION

One of the things that is lacking in the systems we have discussed is the ability to automatically propagate changes through multiple versions of the same system. This notion becomes important when a fundamental change is made to a base system from which multiple different versions have been created. Rather than go through each different version individually and make the required changes, it would be much more efficient to make the change to the base version, and then merge that changed base with each of the different versions, individually, to automatically incorporate the change in each version. This notion could save a tremendous amount of time and effort currently spent by system maintainers to do this. An example of this idea is shown in Figure 4.

Another view of this idea, portrayed in Figure 5, addresses the problem mentioned in the discussion of the Model for Software Maintenance, regarding re-combining parallel genealogies. If two different modifications of a base program contain useful functionality, then automatically integrating these modifications into a program which contains the important aspects of both modifications should be possible.

This thesis is directed towards developing a precisely defined model which shows how this can be accomplished for

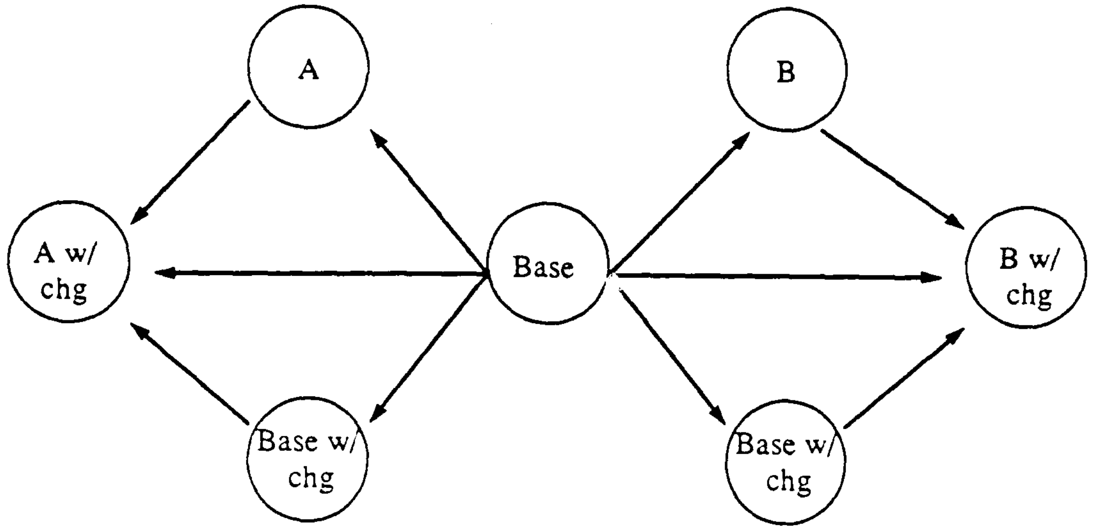

Figure 4. The Notion of Change Propagation Represented Graphically.

PSDL programs. In Chapter II, we review some work previously done on merging pure extensions and integrating modifications. In Chapter III, we take the ideas described in Chapter II and adapt them to PSDL to formulate our model.

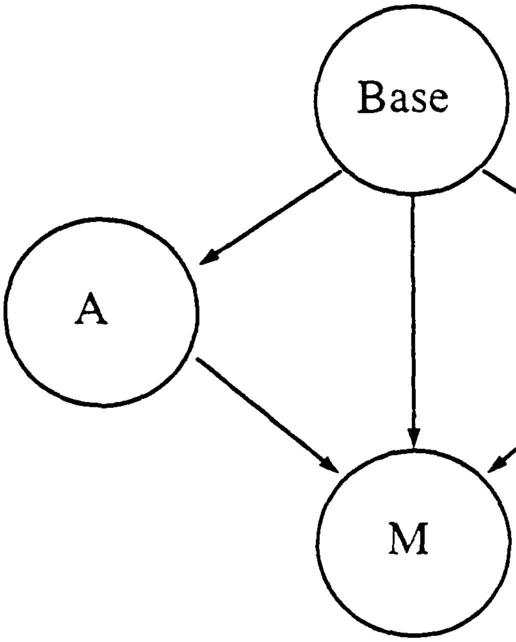

Figure 5. The Idea of Merging Two Different Versions of a Base Program into a Merged Program with the Significant Capabilities of Both Versions is Similar to Propagating Changes.

II. PREVIOUS WORK

This chapter explores some of the work that has been done by other researchers in this area. Section A discusses work on merging software extensions. [Ref. 10] Section B presents an approach to integrating non-interfering software modifications. [Ref. 11]

A. MERGING SOFTWARE EXTENSIONS

1. Extension vs. Modification

Program extensions are additions to the program which extend the domain of the partial function without altering initially defined values. Modifications are additions or changes which do alter initially defined values. In other words, program extensions add functionality to the base program without altering the already existing functionality. For example, consider the program in Figure 6. This program takes two real numbers as input, checks to see if the first is an approximation of the second, and prints "True" if the difference is relatively small and "False" otherwise. This program can fail to produce an output for some inputs, namely y = 0.0. It can easily be extended by adding a filter to check for this possibility, as shown in Figure 7.

Program modifications, on the other hand, change the original functionality of the program. Program Approx_2, shown in Figure 8 is an example of a modification. This

```
program Approx(input, output);
    const
        eps := 0.00001;
    var
        x, y: real;
    begin
        readln(x);
        readln(y);
        if (abs((x - y)/y) ≤ eps)) then
            writeln("True");
        else
            writeln("False");
    end.
```

Figure 6. Example: Program Approx Diverges if y = 0.

```
program Approx_1(input, output);
    const
        eps := 0.00001;
    var
        x, y: real;
    begin
        readln(x);
        readln(y);
        if (y <> 0) then
            if (abs((x - y)/y) ≤ eps)) then
                writeln("True")
            else
                writeln("False")
        else
            if (abs(x - y) ≤ 0) then
                writeln("True")
            else
                writeln("False")
    end.
```

Figure 7. Example: Program Approx_1 is a Compatible Extension of Approx Which is Defined for all Inputs.

program computes a slightly different approximation relation than the program Approx. Since the program no longer provides the original functionality, a modification has occurred.

More formally, using an approximation ordering $\sqsubseteq$, if p is a base program and p $\sqsubseteq$ q, then p approximates q and q is an extension of p. That is to say that q agrees with p everywhere p is defined, and q may be defined in cases where p is not. [Ref. 10]

```
program Approx_2(input, output);
     const
          eps := 0.00001;
     var
          x, y: real;
     begin
          readln(x);
          readln(y);
          if (abs((x - y)) ≤ eps)) then
               writeln("True");
          else
               writeln("False");
     end.
```

Figure 8. Example: Program Approx_2 Uses a Different Approximation Method.

With this ordering in mind, two specifications p and q can be merged by finding the least common extension of p and q, written p $\sqcup$ q, where p and q are base specifications and p $\sqcup$ q is the merged specification. The least common extension is in general not computable, so an approximation is used. [Ref. 10]

2. Definitions

In [Ref. 10], only four domains were considered; specifications, functions, programs, and data types. These four domains are defined in Figure 9. All domains are treated as lattices.[1] The lattice for a domain representing a data type D_0 can be defined as the set $\mathbf{D} = D_0 \cup \{ \perp, \top \}$, where $\perp$ approximates everything and $\top$ is an extension of everything. The definition of the extension relation for **D** is:

$x \sqsubseteq y \Leftrightarrow (\perp \equiv x) \vee (x \equiv y) \vee (y \equiv \top)$[2]

Specification:	Models Intended Behavior
Function:	Implements Actual Behavior
Program:	Algorithms Defining Partial Functions
Data Type:	Set on which Programs Operate

Figure 9. Definitions of Relevant Domains.

[1] Lattices are partially ordered sets with a least upper bound and a greatest lower bound. [Ref. 10]

[2] The strong equality relation $x \equiv y$ results in **True** if x and y are the same element, and **False** otherwise, for all elements of **D** including $\perp$ and $\top$.

In merging extensions, $\perp$ represents an unsuccessful computation, and $\top$ represents the results of combining incompatible[3] data values.

In [Ref. 10] data types are viewed as heterogeneous algebras, where each primitive operation f of the algebra is extended to a full lattice via the following properties:

1. $x_i \equiv \perp \Longrightarrow f(x_1, \ldots, x_i, \ldots, x_n) \equiv \perp$ and
2. $(x_i \equiv \top \ \& \ x_j \not\equiv \perp \text{ for } 1 \leq j \leq n) \Longrightarrow$ $f(x_1, \ldots, x_i, \ldots, x_n) \equiv \top$, for $1 \leq i \leq n$.

Property 1 says that for any element x_i, if $x_i \equiv \perp$ then any operation f with x_i as a parameter returns $\perp$. Property 2 says for any x_i in the parameter list of f, if $x_i \equiv \top$ and no $x_j \equiv \perp$ then f returns $\top$. Conditionals for this domain are extended by the following:

1. (if $\perp$ then x else y) $\equiv \perp$
2. (if $\top$ then x else y) $\equiv \top$.

The domains for functions and specifications are defined as mappings on data type domains, as shown in Figure 10. Orderings for these domains are defined as follows:

1. For $f, g \in$ Func, $f \sqsubseteq g \Leftrightarrow \forall x \in \mathbf{D}[f(x) \sqsubseteq g(x)]$.
2. For $s, t \in$ Spec, $s \sqsubseteq t \Leftrightarrow \forall x \in \mathbf{D}$, $y \in \mathbf{R}[s(x,y) \sqsubseteq t(x,y)]$.

[3] An unsuccessful computation includes infinite computations, and computations which terminate abnormally or with an error message.

$\text{Func} = \mathbf{D} \rightarrow \mathbf{R}$

$\text{Spec} = [\mathbf{D} \times \mathbf{R}] \rightarrow \text{Bool}$

D, **R** are Data Type Domains.

D → **R** is a set of continuous functions with respect to $\sqsubseteq$.

Figure 10. Domains for Functions and Specifications.

The limitation to continuous functions is not considered a "serious restriction", as all computable functions are continuous.[4] [Ref. 10]

$\text{Dom: Spec} \Rightarrow \text{Powerset}[\ \mathbf{D}\ ],$

$\text{Dom}(s) = \{x \in \mathbf{D} \mid \exists\ y \in \mathbf{R}\ [\textbf{True} \sqsubseteq s(x,y)]\}$

$\text{Sat: } [\text{Func} \times \text{Spec}] \rightarrow \text{Bool},$

$\text{Sat}(f,s) \Leftrightarrow \forall\ x \in \text{Dom}(s)\ [\textbf{True} \sqsubseteq s(x,f(x))]$

Figure 11. Definitions of Domain and Satisfy.

Since specifications which leave part of the input space unconstrained are of interest, the definitions shown in Figure 11 are provided to clarify what it means for an input value to be in the domain of a specification and for a function to satisfy a specification. The domain of a

[4] A function f is continuous if and only if $f(\bigsqcup S) = \bigsqcup f(S)$, for all directed sets S. S is directed if and only if every finite subset of S has an upper bound in S.

specification is that set of input values for which an acceptable response can be provided, thus Dom(s) is the set of values of x in the input domain **D** for which there exists an output value y in the output domain **R** . A specification for a function f applied to a pair (x, y) will yield a **True** if y is an acceptable value for f(x), **False** if not, $\perp$ if the specification does not say whether y is acceptable, or $\top$ if the specification is inconsistent with respect to whether y is acceptable or not. $\top$ would be the result if two conflicting specifications were merged. A function f satisfies a specification s if and only if for every value x in the domain of s, f returns a value which is acceptable. Figure 12 shows an example used in [Ref. 10] to clarify this idea. The specification s yields $\perp$ if $x < 0$. In these cases, a correct function can yield any value.

$$s(x,y) = \text{if } 0 \leq x \text{ then } |(y - x^2)| \leq \varepsilon \text{ else } \perp.$$

Figure 12. Example: A Specification for a Square Root Function.

3. Program Merging

The least common extension of two functions is not computable ın general. [Ref. 10] Since the least common extension is the desired result of a merge operation, the theorem shown in Figure 13 is provided.[5]

<u>Theorem</u>: Correctness of Extensions

If s, t are monotonic, $f \sqsubseteq h$, $g \sqsubseteq h$,
Sat(f, s), and Sat(g, t) then Sat(h, ($s \sqcup t$)).

Figure 13. Correctness Theorem. [Ref. 10]

This theorem states that given two monotonic[6] specifications, *s* and *t*, and three functions, *f, g* and *h, if f* satisfies *s* and *g* satisfies *t*, then any common extension *h* of *f* and *g* satisfies the least common extension of *s* and *t*. The function *h* is an approximation for the least common extension of *f* and g, and is sufficient. An approximation can yield an inconsistency in some cases where consistent combinations are possible, but an inconsistency is more acceptable than an undefined or diverging computation, because it can be detected at merge time.

A program consists of a set of function definitions accompanied by an expression. The merging of two such

[5] A proof of the Correctness Theorem can be found in [Ref. 10].

[6] A function f is monotonic if and only if $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.

programs will result in a third program containing a set of functions according to the following rules:

1. Functions which appear in one of the input programs and not in the other will appear in the merged program unchanged.

2. Functions which appear in both of the input programs with the same number of parameters, will be merged.

3. Functions which appear in both of the input programs, but with a different number of parameters, will be merged. The formal parameter list will contain a $\top$ at the place where the inconsistency occurs.

Expressions are merged using normal forms. [Ref. 10] uses rewrite rules to reduce expressions to normal forms. For example, consider the rewrite rules:

1. $y + w \Rightarrow x$
2. $z \Rightarrow w$

Using these rewrite rules on the expression $(x \times (y + z))$ reduces it to the expression $(x \times x)$. Figure 14 provides an example of their use. Normal form merging can be strengthened if axioms and theorems about the data structures are added to the rewrite rules.

Function definitions can be merged using normal forms also. Semantically equivalent function calls are merged by renaming formal parameters in one input version to match the other version, if necessary, and inserting it into the merged program. Even some function calls with the same number (if $x = y + w$ & $w = z$ then $x \times (y + z)$ else $\perp$)

merged with

(if $x = y + w$ & $w = z$ then $x \times (y + w)$ else 0)

yields

(if $x = y + w$ & $w = z$ then $x \times x$ else 0)

Figure 14. Example: A Merge Using Rewrite Rules.

of arguments, but not semantically equivalent can be consistently merged by creating a new function.

The work done in [Ref. 10] was the first of its kind that we were able to find. It provides a mathematical foundation for performing program merges, and shows that merging programs is possible. The next section builds upon this foundation, and provides an algorithm for merging two modifications of a base program.

B. AN ALGORITHM FOR INTEGRATING PROGRAM MODIFICATIONS

Many times in the software evolution cycle, base programs are altered or enhanced in different ways to meet the needs of different customers. This leads to the development of parallel genealogies, as described in Chapter I. At a later time in the evolution cycle, a customer may want a program which has all the capabilities of two different genealogies. This leads to the need for some automated way of integrating two genealogies into a single working program. This automated integration method could also be used to propagate changes made to a base program through all of its offspring. This is accomplished by making the change to the base program, and then integrating that changed base program with each of the offspring.

In [Ref. 11], an algorithm is presented for integrating two non-interfering modifications of a base program. The integration produces a third program which reflects both modifications. This integration method can be useful in recombining parallel genealogies as illustrated above. The algorithm uses program dependence graphs (PDGs) to abstractly represent the programs, then by using program slicing, determines which portions of the two versions are different from the base program. Using this information, the algorithm determines if the changes interfere[7] with each other. If they do not interfere, the different program slices are combined into one integrated PDG, which is then transformed into the final version of the program.

1. Program Dependence Graphs

As mentioned earlier, [Ref. 11] uses PDGs to automate the merging process. A PDG for a program P is a directed graph, G_P with several kinds of vertices connected by

[7] Versions A and B interfere with respect to a Base program if $\exists$ an initial state and variable *x* such that A, B, and Base all compute different values of *x*.

several different kinds of edges. The vertices represent one of the following:

1. The Entry Vertex
2. Initial Definitions: "x := Initial_State(x)"
3. Assignments
4. Control Predicates
5. Final Use Statements: "Final_Use(x)"

Edges represent dependencies between vertices:

1. Control Dependencies: $\Rightarrow_c$
2. Data Dependencies:
    a. Flow Dependence: $\Rightarrow_f$
    b. Def-Order Dependence: $\Rightarrow_{do}$

Control dependence edges, $v_1 \Rightarrow_c v_2$ are contained in G_P if and only if one of the following properties holds:

1. v_1 is an entry vertex and v_2 represents a component of P not subordinate to a control predicate. This type of control dependence edge is always labeled **True**.

2. v_1 is a control predicate and v_2 is a component immediately subordinate to v_1.

    a. If v_1 is a **while** predicate and v_2 is in the loop body, the edge is labeled **True**.

    b. If v_1 is a conditional predicate, the edge is labeled **True** if v_2 is on the **then** branch, **False** if v_2 is on the **else** branch.

Data dependence edges, $v_1 \Rightarrow v_2$ indicate that v_1 must occur before v_2 for data to be valid. G_P contains a flow dependence edge, $v_1 \Rightarrow_f v_2$ if and only if:

1. v_1 defines a variable, say *x*.

2. v_2 uses *x*.

3. Control can reach v_2 after v_1 along a path that doesn't change the value of *x*.

Flow dependence edges due to a particular *x* appear as $(\Rightarrow_f^x)$. Flow dependencies are further classified as loop carried $(\Rightarrow_{lc})$or loop independent$(\Rightarrow_{li})$. v_2 is dependent on v_1 along a loop independent edge, $v_1 \Rightarrow_{li} v_2$, if in addition to a, b and c above, there is an execution path that satisfies c and does not include a backedge to the predicate of the loop that encloses v_2 and v_1. v_2 is dependent on v_1 along a loop carried edge for a loop L, $v_1 \Rightarrow_{lc(L)} v_2$, if in addition to 1, 2, and 3 above, the following properties also hold:

4. There is an execution path which satisfies 3 and includes a backedge to the predicate L.

5. v_1 and v_2 are enclosed in loop L.

Def-order edges, $v_1 \Rightarrow_{do} v_2$, appear in G if and only if the following properties hold:

1. v_1 and v_2 are both assignments to the same variable x.

2. v_1 and v_2 are in the same branch of every conditional statement that encloses them.

3. There is another vertex v_3 that reads x and is dependent on both v_1 and v_2 along flow dependence edges.

4. v_1 occurs before v_2.

Using these components, a program dependence graph can be constructed for any program. Figure 15 shows an example of a simple program and Figure 16 shows its

```
program nada
     sum := 0;
     x := 1;
     while x < 11 do
          sum := sum + x;
          x := x + 1;
     end
end(x, sum)
```

Figure 15. Example: A Simple Program to Illustrate Program Dependence Graphs.

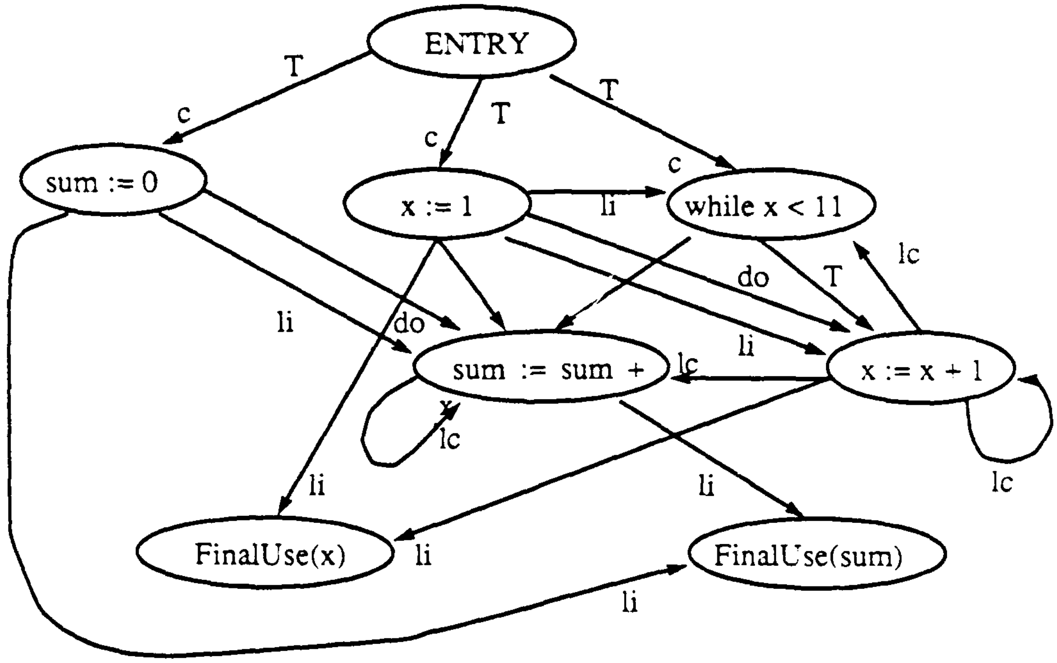

Figure 16. Example: The Program Depend3nce Graph for the Program nada.

associated PDG. By analyzing parts of this graph which affect a certain variable, one is able to observe the effects of a change to the program with respect to that variable. This is done using a technique known as program slicing.

2. Program Slicing

The program slice of a graph G with respect to a vertex s is the subgraph of G containing all vertices which can reach s by way of control or flow dependence edges, along with the edges.

$$V(G/s) = \{w \in V(G) \mid w \Rightarrow_c^* s\}$$

To get the slice of a graph G with respect to one of the output variables, say x, merely take the slice with respect to the vertex labeled "Final_Use(x)". Def-order edges are contained in the slice only if the vertex which depends on it is also included in the slice. This can be extended to a set of vertices $S = \{s_1, s_2, \ldots, s_i\}$ by taking the union of the vertex sets of all of the individual program slices. Figure 17 shows an example of slice of the program **nada** taken with respect to the variable x. Figure 18 shows the corresponding PDG.

3. Program Semantics and Program Dependence Graphs

The problem with text merging, as has been pointed out several times, [Ref. 10,11] is that it discounts the influence of semantics on program structure. In merging programs, the semantics must be merged as well. PDGs provide a way to do that.

```
program nada_x
     x := 1;
     while x < 11 do
          x := x + 1
     end
end(x)
```

Figure 17. Example: The Slice of the Program nada Taken with Respect to x.

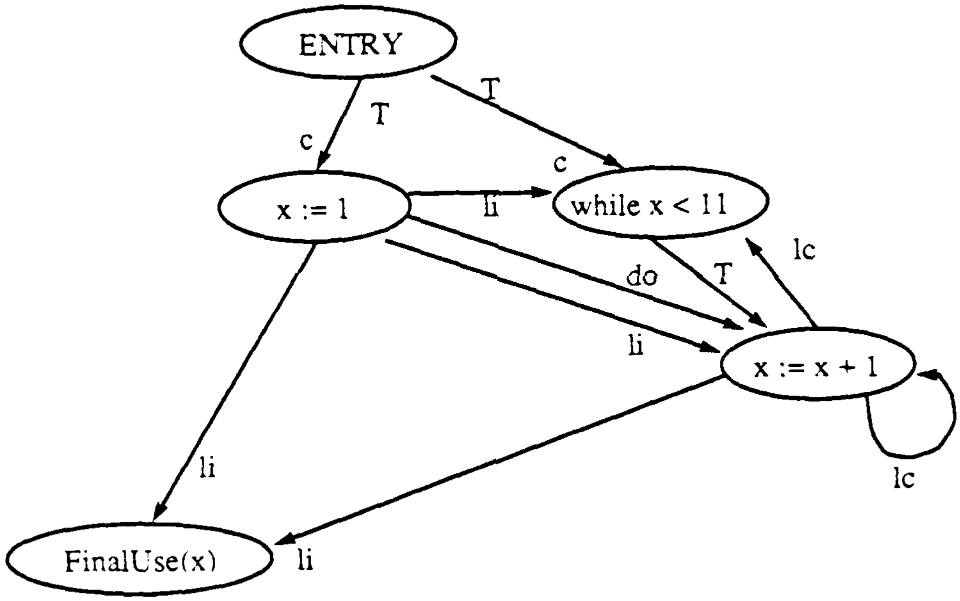

Figure 18. Example: Program Dependence Graph for nada_x.

It can be said that two programs are equivalent if they provide the same results for all possible input values. Two programs P & Q are strongly equivalent, if and only if, *for all possible states* σ, P & Q both diverge when initiated at σ, or they both halt with the same final values. If they are not strongly equivalent, then they are not equivalent. [Ref. 11] provides the theorem shown in Figure 19. This theorem states that if two PDGs are isomorphic then their representative programs are semantically equivalent. The contrapositive to that is inequivalent programs have non-isomorphic PDGs.

Theorem: If P, Q are programs such that G_P is isomorphic to G_Q, then P is strongly equivalent to Q.

Figure 19. Strong Equivalence Theorem.

Another theorem stated in [Ref. 11] is the Slicing Theorem shown in Figure 20. This theorem states that for Q, a slice of program P, P & Q behave equivalently at all places which are common to both P & Q.

Theorem: Let Q be a slice of program P with respect to a set of vertices. If σ is a state on which P halts, then for any state σ' that agrees with σ on all variables for which there are initial-definition vertices in G_Q:

(1) Q halts on σ'.
(2) P and Q compute the same sequence of values at each program point of Q.
(3) The final states agree on all variables for which there are final-use vertices in G_Q.

Figure 20. Slicing Theorem.

4. Determining Behavior Differences in Variants

In order to integrate two different versions of a base program, their differences from the base must be determined. This can be done using the PDGs of the different versions. This model assumes that only changes in the behavior of a modification with respect to its base are significant. If the slice of G_{BASE} with respect to a vertex *v* is different from the slice of *GA*, where A is the modification, with respect to *v*, then this is an indication of possible changed behavior. All such vertices where G_{BASE} and G_A differ are called the *affected points* $AP_{A,BASE}$ of G_A. The slice $G_A/AP_{A,BASE}$ is a graph which captures the behavior of A that is different from the BASE. Determining the *affected points* by checking the program slices for every vertex in the graph is not very efficient. [Ref. 11] presents a function for doing this that requires at most two complete examinations of the graph. It is the function called *AffectedPoints*[Ref. 11, P. 21], and is based on the following three observations:

1. All vertices in G_A but not in G_{BASE} are affected points.

2. Each vertex *w* of G_A with a different set of incoming flow or control edges than in G_{BASE} gives rise to a set of affected points, namely the vertices which can be reached via zero or more flow or control edges from *w*.

3. Each vertex *w* of G_A with an incoming def-order edge, due to a vertex *u* that does not appear in G_{BASE}, gives rise to a set of affected points, namely the vertices that can be reached via zero or more flow or control edges from *u*.

5. Merging Program Dependence Graphs

Once the PDGs have been created for the different modifications and the base, and all the differences between the modifications and the base have been determined, the merged PDG G_M must be created. G_M is formed by taking the union of the slices representing the changed behaviors of versions A and B with respect to the base and the s ice representing the preserved behavior of the base in both versions A and B. This final slice contains the subset of $V(G_{BASE})$ for which the slices in all three versions are isomorphic, and is represented by $PP_{BASE,A,B}$.

1. $PP_{BASE,A,B} = \{v \in V(G_{BASE}) \mid (G_{BASE}/v) = (G_A/v) = (G_B/v)\}$
2. $G_M = (G_A/AP_{A,BASE}) \cup (G_B/AP_{B,BASE}) \cup (G_{BASE}/PP_{BASE,A,B})$

6. Determining Interference Between Modifications

A merged PDG created as described above can inaccurately reflect the changed behavior of the two modifications in two ways. First, the union of two feasible[8] PDGs is not necessarily a feasible PDG. Secondly, G_M may not preserve the differences in the behavior of the modifications with respect to the base. Interference when either of these conditions occurs. Testing for interference due to the first condition is done during the program reconstitution process. A theorem is provided in Reference 5 which states that the

[8] A PDG G_P is feasible if it is a PDG for a program P. The slice G/S is feasible if $S \subseteq V(G)$ and G is feasible.

function ReconstituteProgram will succeed if and only if G_M is feasible. Testing for interference due to the second condition can be done by merely checking the following condition:

$$G_M/AP_{A,BASE} = G_A/AP_{A,BASE} \text{ and } G_M/AP_{B,BASE} = G_B/AP_{B,BASE}$$

7. Program Reconstitution

Program reconstitution is accomplished through the use of a function called ReconstituteProgram. This function first attempts to order the tree created by the control dependencies between the vertices using the flow dependencies. It then transforms the graph into an abstract syntax tree from which a program is formed. A PDG is then created for the program created and is checked against G_M. This function will fail if either the vertices cannot be ordered or the PDG of the created program is not isomorphic to G_M.

The algorithm Integrate described in this section is far from perfect, but up to this time, seems to be the most complete work of its kind. Portions of the algorithm still have problems, though. One example is that the ordering of the vertices done in Step 5 of the algorithm is a problem that is NP-Complete.

C. SUMMARY

In this chapter, we have explored two different approaches to the software integration problem. In Section A, work done in modelling the merging of pure extensions of a program was presented. This provided a sound foundation for understanding the work done in Section B. Section B explored the integration of non-interfering versions of programs through the use of program dependence graphs and an algorithm for integrating different slices of the graphs for different versions into a merged graph, from which a merged program can be created. These two bodies of work have helped significantly with our understanding of the complexities of program integration and the development of a model to account for those complexities.

III. A MODEL FOR MERGING PSDL PROGRAMS

PSDL programs are made from the combination of one or more Abstract Data Types and/or Operators. Since, for the current implementation of PSDL, all Abstract Data Types are implemented in Ada, merging them is not within the scope of this thesis and is not included in our model. We do model the merging of a base PSDL operator, Base, with two modifications or extensions, A and B, of Base, into a least common extension, M. We refer to this three-way merging model as "change-merging", to prevent confusion with other merging models. This chapter defines the operations, $\sqcap$, $\sqcup$, and $-$, and the relation $\sqsubseteq$ as they pertain to change-merging A, Base and B into M. As was pointed out in Chapter 2, [Ref. 10] tells us that the least common extension of two programs is not computable in the general case. This fact is also true with PSDL programs. We will show that an approximation to the least common extension is enough to provide a successful change-merge in most cases.

The relation $\sqsubseteq$ is defined as the approximation relation for the lattice created by combining all PSDL operators together with a $\top$ and a $\bot$ as shown in Figure 21. If Y is an extension of X, then we say X approximates Y, written X $\sqsubseteq$ Y. The $\top$, or top element, is an extension of all possible PSDL operators. This is an overconstrained artificial component which represents an inconsistency. The $\perp$, or bottom element, is an approximation of all possible PSDL operators. This is an artificial unconstrained component that represents an undefined element. Having these components in the lattice provides a way for us to completely define a change-merge operation on PSDL operators. If $\top$ is the result of a change-merge operation, there is no consistent way to provide a change-merge that is syntactically correct. $\perp$, on the other hand, is a component that is an undefined (i.e. an unimplemented or non-terminating program).

The difference between two PSDL operators, A - B, gives the behavior found in A and not in B. Any functionality which is common to both operators is not included in A - B. The greatest common approximation of two PSDL operators, A and B, is written A $\sqcap$ B. The greatest common approximation of three operators, A $\sqcap$ Base $\sqcap$ B gives us the behavior which is common to all three operators.

To get the desired behavior of M we must combine this common behavior, A $\sqcap$ Base $\sqcap$ B, with the difference between the behavior of Base and each of the two modifications or extensions, A - Base and B - Base. From this we can conclude that change-merging the three versions can be done using the following formula:

$$M = A[Base]B = (A \sqcap Base \sqcap B) \sqcup A - Base \sqcup B - Base.$$

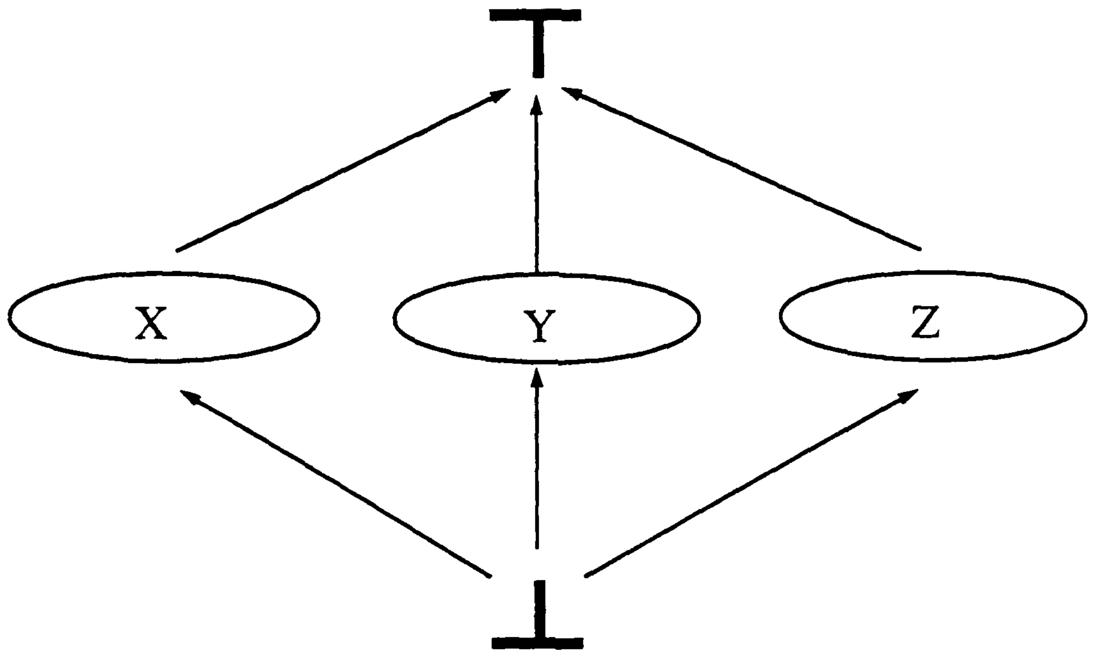

Figure 21. The Set of All PSDL Operators Forms a Lattice.

It turns out that it is not necessary to use the greatest common approximation of all three operators. The difference between $A \sqcap B$ and $A \sqcap \text{Base} \sqcap B$ is contained in the modifications A - Base and B - Base, as shown by the following equations and inequalities:

$$\begin{aligned}&(A \sqcap B) - (A \sqcap \text{Base} \sqcap B) = (A \sqcap B) - \text{Base} = \\ &\quad (A - \text{Base}) \sqcap (B - \text{Base}) \sqsubseteq \\ &\qquad (A - \text{Base}) \sqcup (B - \text{Base})\end{aligned}$$

Thus in merging the three versions, it is sufficient to merge the greatest common approximation of A and B with the behavior differences between the two modifications and the Base, as shown in the following:

$$M = A[Base]B = (A \sqcap B) \sqcup (A - Base) \sqcup (B - Base).$$

This chapter defines what this equation means in change-merging the different components of a PSDL operator.

PSDL operators consist of two major parts: a specification and an implementation.[1] The specification of an operator A, denoted S_A[2], defines the external interfaces of the operator and identifies its functionality through the use of text descriptions and keywords. The change-merging of two PSDL specifications is discussed in Section A. The implementation part of an operator A provides an Enhanced Data Flow Diagram consisting of a data flow diagram, a set of internal data streams, and a set of constraints which control the internal operations of the operator. For this model, we separate the implementation part of the operator into two separate and distinct problems. The first is change-merging two simple data flow diagrams, denoted D_A and D_B, discussed in Section B, and the second is the integration of two sets of internal data streams, denoted DS_A and DS_B, and control constraints, denoted C_A and C_B, discussed in Section C.

[1] The change-merging of specifications and implementations is different, so we handle it separately.

[2] In our model, the subscript of a set signifies which operator it represents.

A. SPECIFICATIONS

The specification of a PSDL operator, A, tells the rest of the program what the operator does. Changes to the specification of an operator ptoentially affect all of the other parts of the program where the operator is used. We assume the author of a change to an operator specification has also made the necessary changes in all of the contexts where the operator is used. For this reason, change-merging operations must be applied to entire prototypes. However, changes to entire programs are merged by merging changes to corresponding sub-components. This section focuses on the change-merge operation for the specification of each sub-component.

1. Interfaces

The interface of a PSDL operator is the definition of the operator's external contacts. It contains the input set expected by the program, I_A, the output set that can be expected, O_A, and the set of generic parameters that may be instantiated, GN_A. I_A, O_A, and GN_A are all ordered sets. It also contains a set of internal state variables, St_A, a set of possible exceptions, E_A, and a set of timing requirements that are met by the program, T_A. St_A, E_A, and T_A are all unordered sets.

a. *Ordered Sets*

Ordered sets are a significant building block for many programming languages, including PSDL. For the purpose of change-merge operations, ordered sets should be modelled using a flat lattice, as shown in Figure 22. When the order of a set is significant, then any change made to the set is an incompatible change and creates a set which is neither an approximation nor an extension of the original set. This means that the only set which is an approximation for the ordered set is the undefined set, signified by the ⊥ in Figure 22, and the only set which is a compatible extension of the ordered set is the overconstrained set signified by the ⊤ in Figure 22. The applications of this structure to PSDL specifications are explained next.

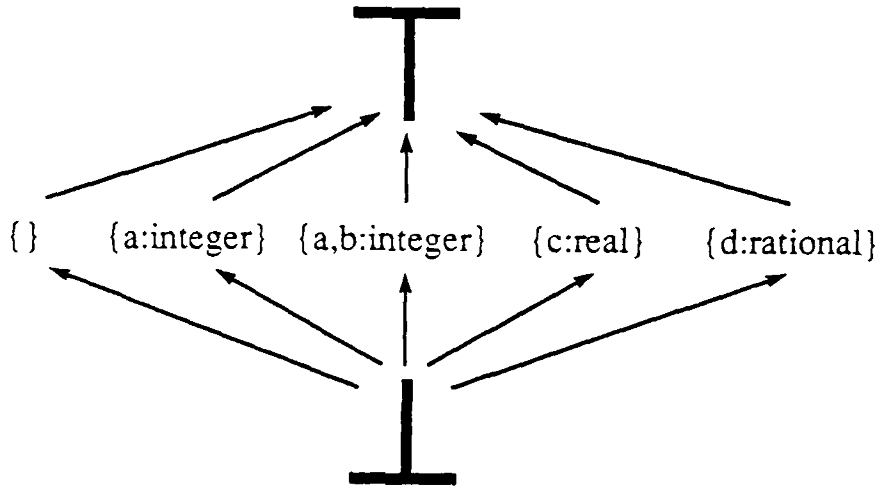

Figure 22. Ordered Sets Have a Flat Lattice Structure.

(1) Input and Output[3]

Input and Output interfaces are sets of input and output streams. The order of these sets is significant because actual parameters are associated with formal parameters based on the order in which they appear. In change-merging I_A, I_B and I_{Base} into I_M, any change between the interface set of Base and the two modified versions is significant, and must be preserved in the change-merged version. The change-merged set of inputs, or outputs, is determined by the following rule:

$$I_M = I_A[I_{Base}]I_B = (I_A - I_{Base}) \sqcup (I_A \sqcap I_B) \sqcup (I_B - I_{Base})$$

Based on this rule, one of the following three situations can occur:

1. If both of the modifications have the same interface set as the base, then: $I_M = \bot \sqcup I_{Base} \sqcup \bot = I_{Base}$.
2. If one of the two modifications, say I_A, is the same as the base, and the other is not, then: $I_M = \bot \sqcup \bot \sqcup I_B = I_B$.
3. If both of the modifications are different from the base version, then: $I_M = I_A \sqcup \bot \sqcup I_B = \top$.

The first situation is the case in which no changes were made between the inputs of the Base and the two modifications. In this case, the change-merged version

[3] The change-merging of input and output sets is identical, so only the change-merging of input sets is shown here.

should have all of the same inputs, or outputs. The second situation is the case in which only one of the modifications changed from the base. In this case, the change from the base is significant and must be preserved in the change-merged version. The third situation is the case where both of the modifications changed from the base. The result is a conflict because there is no proper PSDL specification that is consistent with both modifications.

An example of an interface change-merge is shown in Figure 23. In this example, $I_A \not\equiv I_{Base}$, but $I_B \equiv I_{Base}$, so $I_M \equiv I_A$. $O_{Base} \not\equiv O_A \not\equiv O_B$, so $O_M \equiv \top$.

S_{Base} = INPUT

```
     x: integer
     y: real
   OUTPUT
     w: integer
     z: string
```

S_A = INPUT

```
     x: integer
   OUTPUT
     w: integer
     t: integer
     z: string
```

S_B = INPUT

```
     x: integer
     y: real
   OUTPUT
     w: integer
```

S_M = INPUT

```
     x: integer
   OUTPUT
```

$\top$

Figure 23. Example: An Interface Merge.

(2) Generic Parameters

The Generic interface is contained only in template operators. Template operators are operators in the Software Base used to instantiate software components. Change-merging generic parameters is similar to change-merging input and output parameters with the exception that in addition to value parameters, generic parameter sets may also contain operator parameters and type parameters. Changes to generic sets will follow the same rules as Input and Output sets. Figure 24 shows an example of a change-merge operation on generic parameters.

b. Unordered Sets

Unordered sets are modelled using a "Powerset Lattice"[4], as shown in Figure 25. Because unordered sets are modelled using this type of lattice, more freedom can be exercised in change-merging them. Change-merge operations do not follow the same rules for these unordered sets as for ordered sets.

(1) States

State variables differ from input and output variables in that, abstractly, they are tuples, containing a name, a type and an initial value. As the set of state variables is unordered and invisible to the rest of the program, the state set can be increased or decreased without

[4] A "Powerset Lattice" is a lattice whose ordering is based the powersets of a given data type. A is a compatible extension of B, if A is a subset of B.

```
$GN_{Base}$ = GENERIC
            t1: type,
            t2: type,
            o1: operation[i1,i2:t1,o1:t2],
            v1: integer

$GN_{A}$    = GENERIC
            t1: type,
            t3: type,
            o2: operation[i1:t1,o1:t3],
            v1: integer

$GN_{B}$    = GENERIC
            t1: type,
            t2: type,
            o1: operation[i1,i2:t1,o1:t2],
            v1: integer

$GN_{M}$    = GENERIC
            t1: type
            t3: type
            t4: type
            o2[i1:t1,o1:t3],
            v1: integer
```

Figure 24. Example: A Merge of Generic Parameters.

affecting other parts of the program. In change-merging state variable sets, the operations $\sqcap$, $\sqcup$, and - are equivalent to the corresponding set operations, $\cup$, $\cap$ and -. The third part of the tuple, the initial value, requires an additional check in the change-merging process. These initial values are ordered using a flat lattice, because they are ordinary data values. The initial value of a change-merged state variable follows the same change-merging rules as input and output variables. If all three versions have different initial values for the same state variable, then the change-merged

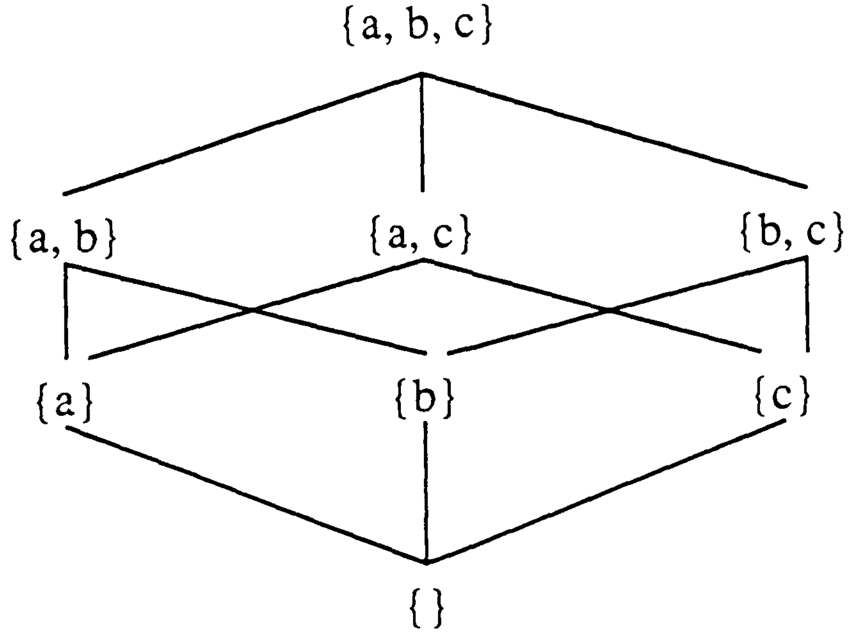

Figure 25. Unordered Sets Have a Powerset Lattice Structure.

version will contain a $\top$ in the place where the initial value is assigned. If only one of the modifications assigns a different initial value than the base version, then the change-merged version will contain the initial value of the one that was different. Figure 26 shows an example of change-merging state variable interfaces. In this example, the state variable *s* is assigned a value in Base which is changed by version A, but not B.

(2) Exceptions

The exceptions interface is a list of identifiers which denote exception values which may be returned by the operator. Consequently $\sqcup$, $\sqcap$, and - can be

St_{Base} =
STATES
  s: integer initially 0
  t: real initially 30.0

St_A =
STATES
  r: natural initially 10
  s: integer initially 5
  t: real initially 10.0

St_B =
STATES
  p: integer initially 0
  s: integer initially 0
  t: real initially 20.0

St_M =
STATES
  p: integer initially 0
  r: natural initially 10
  s: integer initially 5
  t: real initially $\top$

Figure 26. Example: A Merge of State Variable Interfaces.

interpreted as the corresponding set operations, $\cup$, $\cap$, -. Exceptions which appear in one or both of the modified versions, and not in the base, will appear in the change-merged program. Exceptions which appear in the base and do not appear in at least one of the modifications will not appear in the change-merged program. The following formula defines the exception set of the change-merged program, E_M:

$$E_M = [E_A \cap E_B] \cup [E_A - E_{Base}] \cup [E_B - E_{Base}]$$

The expression $E_A \cap E_B$ yields the set of exceptions which are common to all three versions. The expression $E_A - E_{Base}$ yields the set of exceptions which are in modification A and not in the base. The expression $E_B - E_{Base}$ yields the set of exceptions which are in modification B and not in the base. These sets are added together to form E_M. An example of a change-merge on exception sets is found in Figure 27.

E_{Base} = EXCEPTIONS
EXCEPTION1,
EXCEPTION2,
EXCEPTION3

E_A = EXCEPTIONS
EXCEPTION1,
EXCEPTION2,
EXCEPTION5

E_B = EXCEPTIONS
EXCEPTION1,
EXCEPTION3,
EXCEPTION4

E_M = EXCEPTIONS
EXCEPTION1,
EXCEPTION4,
EXCEPTION5

Figure 27. Example: Merging Exception Sets.

(3) Timing Information

There are three different types of timing information found in specifications of PSDL operators, Maximum Execution Time(MET), Maximum Response Time(MRT), and Minimum Calling Period(MCP). MET is the maximum CPU time that an operator can use to perform its assigned task. MRT is the maximum amount of real time between the arrival of an input value on the input stream and the placement of an output value on the output stream. MCP is the minimum amount of time between invocations of an operator.

Change-merging MET, MRT and MCP timing information is done in much the same way as state variable definitions. If the timing information is unchanged in both modifications from the base, then it will remain the same in the change-merged version. If it is the same in one of the modifications as the base, but different in the other, then the changed value will be the value assigned to the change-merged version. If all three versions contain a different timing value, then the change-merged version will contain a timing value of $\top$. Examples are shown in Figure 28.

$$\text{MET } 20 \; [\; \text{MET } 20 \;]\; \text{MET } 20 \equiv \text{MET } 20$$

$$\text{MCP } 40 \; [\; \text{MCP } 40 \;]\; \text{MCP } 30 \equiv \text{MCP } 30$$

$$\text{MRT } 10 \; [\; \text{MRT } 20 \;]\; \text{MRT } 30 \equiv \text{MRT } \top$$

Figure 28. Examples: Merging Timing Information.

2. Functionality

The functionality of an operator specification is what differentiates it from other operators. Through the use of keywords, the operator can be distinguished from other operators in the database during the retrieval process. Text descriptions are provided for use by the engineer. Axiomatic descriptions are provided to allow expert system techniques to be used in the retrieval process.

a. *Keywords*

The keywords section of an operator specification is a set of distinguishable words which explicitly identifies the functionality of the operator. The order of these words is not significant. The change-merging of the set of keywords is done in the same way as the set of exceptions.

b. *Informal Description*

The informal description of a PSDL operator is a textual explanation of the functionality of the operator. It has no formalized sub-structure, therefore, the accurate change-merging of textual explanations, is not within the scope of this thesis. It is assumed that accurate change-merging of the informal description will be done by the engineer overlooking the change-merge operation.

c. *Formal Description*

The formal description of the PSDL operator provides an axiomatic representation of the functionality of the operator. Mathematical properties can potentially be automatically change-merged using currently available technology, but as this portion of the language has not yet been determined in detail, it is impossible for us to provide a method at this time.

B. DATA FLOW DIAGRAMS

A PSDL Data Flow Diagram, for an operator A, is a graph D_A = {O, L}, where O is a set of vertices which represent the component operators of A, including the constant operator EXT representing external contacts, and where **L** is a set of links(labelled edges) which represent the data streams entering and leaving the elements of **O**. The labels for the links are the names of the data streams they represent.

Between any two vertices, o_1 and o_2, there is an edge for each data st eam, *x*, that is an output of o_1 and an input for o_2. A data flow diagram can have parallel edges, since operators can have multiple outputs and/or multiple inputs. Figure 29 shows three examples of PSDL Data Flow Diagrams:

1. An operator with no inputs or outputs.
2. An operator with one input and one output.
3. A composite operator with multiple data streams between its two component operators and multiple output streams.

We define the change-merging operations on PSDL data flow diagrams in terms of a bipartite graph $\mathbf{B}_A$ = {**V**, **S**, **LI**, **LO**}, where **V** is the set of operators in $\mathbf{D}_A$, **S** is a set of vertices which represent the data streams of operator A, **LI** is a set of edges from a stream vertex to an operator vertex, representing input links, and **LO** is a set of edges from an operator vertex to a stream vertex, representing output links. According to this model, the data flow diagrams in Figure 29 have the following bipartite graph representations:

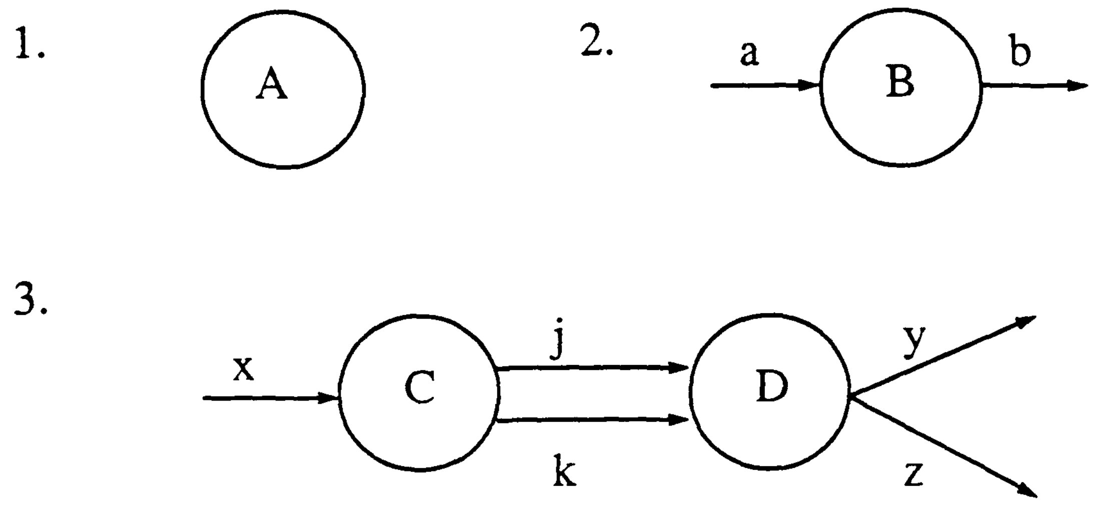

Figure 29. Example: PSDL Operator Data Flow Diagrams

```
1.  G.V = {A,EXT}
    G.S = {}
    G.LI = {}
    G.LO = {}

2.  G.V = {B,EXT}
    G.S = {a,b}
    G.LI = {(a,B),(b,EXT)}
    G.LO = {(EXT,a),(B,b)}

3.  G.V = {C,D,EXT}
    G.S = {x,j,k,y,z}
    G.LI = {(x,C),(j,D),(k,D),(y,EXT),(z,EXT)}
    G.LO = {(EXT,x),(C,j),(C,k),(D,y),(D,z)}
```

Figure 30 gives graphical illustrations of these bipartite graphs.

The bipartite graph models have no edges which link an operator vertex with another operator vertex, or a stream

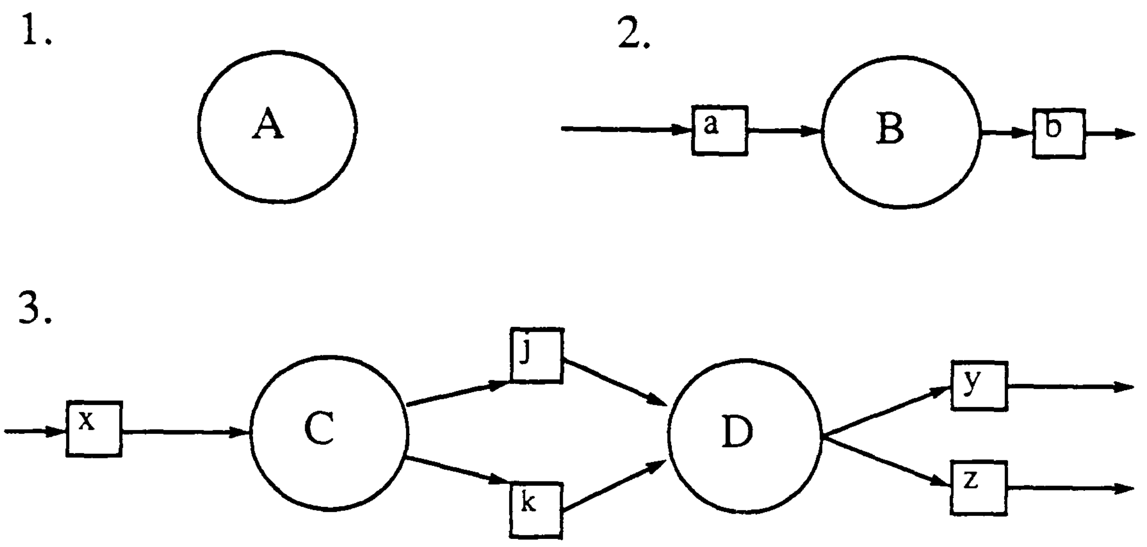

Figure 30. Example: Corresponding Bipartite Graphs for Examples in Figure 29.

vertex with another stream vertex. For each input to an operator vertex, v_1, there is an edge in **LI** which flows from a stream vertex s_1. For each output from an operator vertex v_1, there is an edge in **LO** which flows to a stream s_2.

Change-merging the data flow diagrams is done by change-merging the graphs $\mathbf{G_{Base}}$, $\mathbf{G_A}$ and $\mathbf{G_B}$ by subsets **V**, **S**, **LI** and **LO**. The operations $\sqcup$, $\sqcap$, and - can be interpreted as the corresponding operations $\cup$, $\cap$, and -. The following equation defines the way this change-merge is accomplished:

$$\mathbf{G_M = [G_A - G_{Base}] \sqcup [G_A \sqcap G_B] \sqcup [G_B - G_{Base}]}$$

The greatest common approximation is obtained for the Base and the two modifications by taking the intersection on all components of the graph. Then these common components are added to the disjoint components of each modification by subtracting out the parts of the two modifications which are also in the base. This operation preserves the functionality common to all the versions, while ensuring that significant changes made by the two modifications are included in the change-merged graph. An example of change-merging operations on bipartite graphs is shown in Figure 31, and illustrated graphically in Figure 32.

Base.**V** = {A,EXT}
Base.*S* = {x,y}
Base.**LI** = { (x,A), (y,EXT) }
Base.**LO** = { (EXT,x), (A,y) }

A.**V** = {A,A1,EXT}
A.*S* = {x,y}
A.**LI** = { (x,A), (x,A1), (x,y,EXT) }
A.**LO** = { (EXT,x), (A,y), (A1,y) }

B.**V** = {A,B,EXT}
B.*S* = {x,y,t}
B.**LI** = { (x,A), (y,B), (t,EXT) }
B.**LO** = { (EXT,x), (A,y), (B,t) }

M.**V** = {A,A1,B,EXT}
M.**S** = {x,y,t}
M.**LI** = { (x,A), (x,A1), (y,B), (t,EXT) }
M.**LO** = { (EXT,x), (A,y), (A1,y), (B,t) }

Figure 31. Example: Merge Operation on Data Flow Diagrams.

In this example, the change-merged set of operator vertices is obtained by adding together the operator vertices which are common to all three versions with the operator

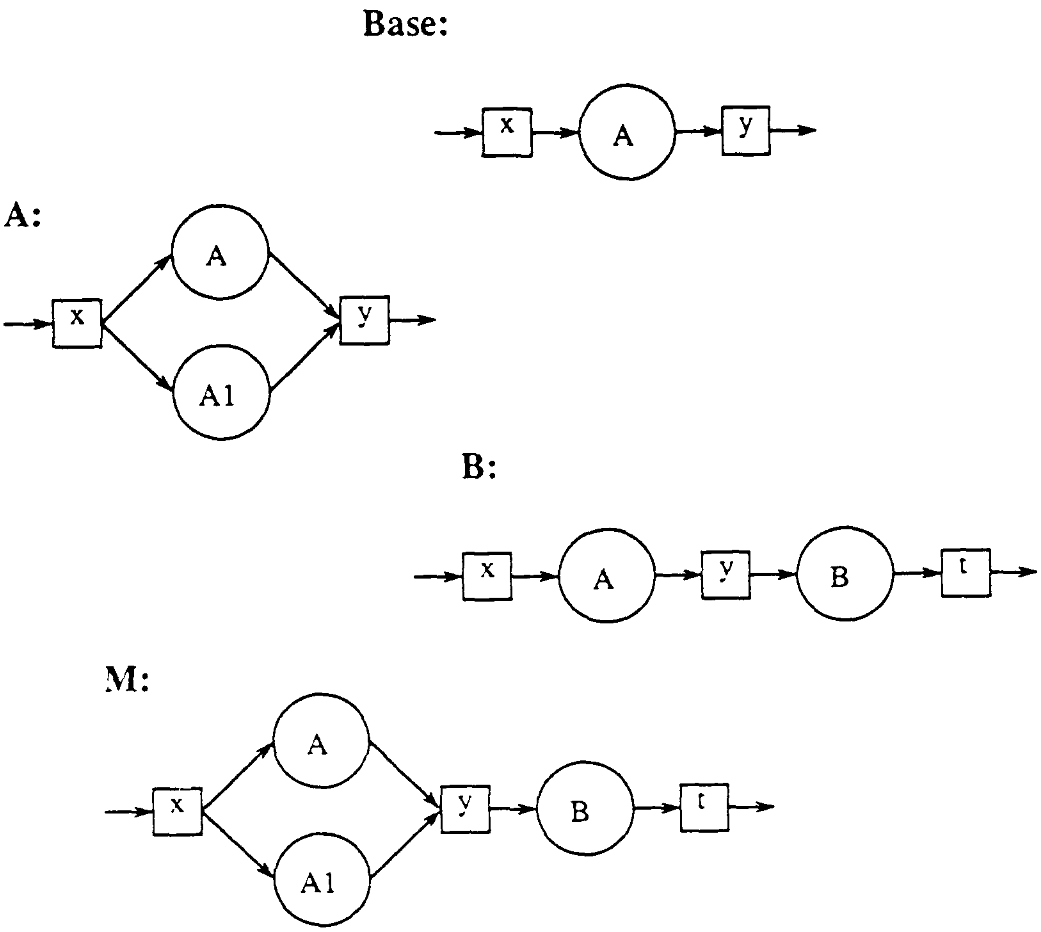

Figure 32. Graphical Illustrations for the Merging Operation in Figure 31.

vertices which are particular to the modifications. The sets of stream vertices and the sets o: edges are treated similarly.

C. DATA STREAMS AND CONTROL CONSTRAINTS

1. Data Streams

A set of data stream definitions, DS_A defines variables which exist internally to the operator, A, and are not defined in the specification. The order in which the declarations appear is not significant. They have the same structure as exception declarations, and can be change-merged using the same rules. If a stream appears in DS_{Base}, then it appears in DS_M if and only if it appears in both DS_A and DS_B. If a stream does not appear in DS_{Base}, then it appears in DS_M if and only if it appears in at least one of the sets DS_A and DS_B.

1. $x \in DS_{Base} \wedge x \in DS_A \wedge x \in DS_B \Rightarrow x \in DS_M$.
2. $x \in DS_{Base} \wedge \neg(x \in DS_A \wedge x \in DS_B) \Rightarrow \neg(x \in DS_M)$.
3. $\neg(x \in DS_{Base}) \wedge (x \in DS_A \vee x \in DS_B) \Rightarrow x \in DS_M$.
4. $\neg(x \in DS_{Base}) \wedge \neg(x \in DS_A \vee x \in DS_B) \Rightarrow \neg(x \in DS_M)$.

2. Control Constraints

Control constraints are a set of pre-conditions which control the firing of particular components, and post-conditions which determine the output provided by those components. The control constraints appear in the change-merged operator according to the same rules as the data stream definitions. Any control constraint which appears in all three input versions in the exact same way will appear in the change-merged operator without change. Any operator which appears in one or both of the modifications, but not in the base, will appear unchanged as long as the conditions of the constraint are the same. Changes in conditions are handled differently depending on the type of constraint. Input and output guards, and conditional exceptions, "TRIGGERED IF", "OUTPUT IF", "EXCEPTION IF", and timer operations have logical predicates as conditions. Timer operations are not change-merged as straight-forwardly as other predicate c nstraints. Different operations exist for different activities. Start, stop, read, and reset are the four timer operations used in PSDL. The read operation has no effect on the state of the timer, so these operations can be merged independently. If a read operation appears in all three versions, or appears in at least one of the modifcations, but not in the base, then it appears in the change-merged version as well. The other timer operations do affect the state of the timer. The start and stop operations affect the run state of the timer, and the reset operation affects the value state of the timer. The reset operation is thus independent of the others, and hence cna be merged according to the same rules as the read operation. The start and stop operations must be change-merged using a flat lattice ordering relation, as with inputs and outputs. This lattice is shown in Figure 33. The predicates which accompany the control constraints are change-merged according to the usual rule, $A[Base]B = (A - Base) \sqcup (A \sqcap B) \sqcup (B - Base)$, where the operations $\sqcup$, $\sqcap$, and $-$ are interpreted as follows:

1. $a \sqcup b \Rightarrow a \vee b$

2. $a \sqcap b \Rightarrow a \wedge b$

3. $a - b \Rightarrow a \wedge \neg b$

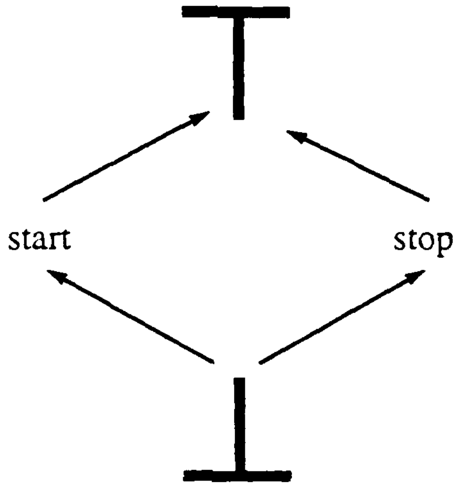

Figure 33. Start and Stop Timer Operations are Merged Using a Flat Lattice Ordering Relation.

"PERIOD" and "FINISH WITHIN" have integer values as conditions. These values are ordered using a flat lattice and can be change-merged in the same way as described for Maximum Response Time and Minimum Calling Period. An example of a control constraint change-merge is shown in Figure 34. In this example, the constraint on A2 does not appear in M because it appeared in Base and B, but not in A. The constraints on A3 and A4 appear in M because they are not in the base and do appear in one of the modifications. The predicate for the constraint on A1 is different in A and B, so according to the rule above, the result of the merge is calculated as follows:

$$\begin{aligned} & y < 0 \; [\; y \leq 0 \;]\; y \geq 0 = \\ & \quad (y < 0 \wedge \neg(y \leq 0)) \vee (y < 0 \wedge y \geq 0) \vee (y \geq 0 \wedge \neg(y \leq 0) \\ = & \quad \textbf{False} \vee \textbf{False} \vee (y \geq 0 \wedge (y > 0) \\ = & \quad (y > 0). \end{aligned}$$

The "READ" operation appeared in the change-merged version, because it appeared in one of the modifications. the other timer operation did not appear in the merged version, because it did appear in the base and in one of the modifications, but not in the other.

C_{Base} = CONTROL CONSTRAINTS

```
A1 TRIGGERED IF y ≤ 0
A2 TRIGGERED BY SOME x
START TIMER2 IF TRUE
READ TIMER1 IF z < 0.01
```

C_A =

```
CONTROL CONSTRAINTS
  A1 TRIGGERED IF y ≥ 0
  A3 PERIOD 30 ms
```

C_B =

```
CONTROL CONSTRAINTS
  A1 TRIGGERED IF y < 0
  A2 TRIGGERED BY SOME x
  A4 FINISH WITHIN 20 ms
  START TIMER2 IF TRUE
```

C_M = CONTROL CONSTRAINTS

```
A1 TRIGGERED IF y > 0
A3 PERIOD 30 ms
A4 FINISH WITHIN 20 ms
READ TIMER1 IF z < 0.01
```

Figure 34. Example: A Merge of Control Constraints.

D. CORRECTNESS OF CHANGE-MERGE OPERATION

Thus far, we have described a model which provides for separate merging of specifications and implementations. We have demonstrated that for the individual cases, the model either produces a program which is syntactically correct or provides evidence to indicate where an inconsistency exists. There are still two things that need to be shown. First, we must show that the change-merged implementation is consistent and correctly implements the change-merged specification. Sub-section 1, below, provides a boolean function ***Imp***(Graph, Specification) which is true if the interface of Graph corresponds to Specification, and a theorem which defines the necessary conditions for a consistent change-merge. Secondly, we must discuss the semantic properties of our model. These are discussed in sub-section 2, below.

1. Consistency of Model

In developing our model for the change-merge operation, we chose to handle the specifications and implementations separately. This is beneficial for developing the change-merge model, but requires showing that the implementation produced by the change-merge correctly implements the change-merged specification. Consistency between an implementation and a specification is defined as follows:

Imp: PSDL Graph × PSDL Spec $\Rightarrow$ Bool

$$
\begin{aligned}
&\textbf{\textit{Imp}}(G_A,\ Sp_A) \Leftrightarrow \\
&\quad x \in I_A \Leftrightarrow (x \in A.S \wedge (EXT,\ x) \in A.LI) \wedge \\
&\quad y \in O_A \Leftrightarrow (y \in A.S \wedge (y,\ EXT) \in A.LO) \wedge \\
&\quad z \in St_A \Rightarrow (z \in DS_A \wedge \exists w \in A.V\ [(w,\ z) \in A.LI] \\
&\qquad\qquad \wedge \exists v \in A.V\ [(z,\ v) \in A.LO])
\end{aligned}
$$

This definition states that the graph G_A correctly implements the specification Sp_A if and only if all of the following are true:

1. A *x* will appear in the input set of the specification if and only if it appears in the set of streams of the graph, and there is an edge in the set of input links from EXT to *x*.

2. A *y* will appear in the output set of the specification if and only if it appears in the set of streams of the graph, and there is an edge in the set of output links from *y* to EXT.

3. A *z* will appear in the set of states in the specification if and only if it appears in the set of data streams of the operator, and there is an operator, *w*, in the set of vertices of the graph such that there is an edge from *w* to *z* in the set of input links, and there is an operator, *v*, in the set of vertices of the graph such that there is an edge in the set of output links from *z* to *v*.

To show that the change-merged implementation is consistent with its specification, we provide the theorem in Figure 35. This theorem shows that if the three input versions are consistent, the change-merged version will be consistent. Our proof of this theorem is contained in Appendix B.

2. Semantic Properties of the Model

The least common extension of two programs provides the desired semantics for a merging operation, but as was

THEOREM: If Sp_A, Sp_{Base} and Sp_B are PSDL operator specifications, B_A, B_{Base} and B_B are operator graphs, ***Imp***(B_A, Sp_A), ***Imp***(B_{Base}, Sp_{Base}) and ***Imp***(B_B, Sp_B), then ***Imp***$(B_A[B_{Base}]B_B, Sp_A[Sp_{Base}]Sp_B)$.

Figure 35. Consistency of Change-merge.

pointed out in [Ref. 10], this is not computable in the general case. We write the partial function computed by an operator implementation as $F(G_A)$. Thus, the partial function computed by the resultant implementation of our change-merge is $F(G_M)$. The result of a semantic merge on the three partial functions, $F(G_A)[F(G_{Base})]F(G_B)$ is written as F_M. The ideal result for our model is: $F(G_M) = F_M$. This ideal cannot always be realized in practice, because F_M is not computable in the general case.

The best result that may be practically realizable is: $F_M \sqsubseteq F(G_M)$. In this situation, the change-merge operation produces a result which is compatible with the ideal result, but which may contain inconsistent values, $\top$, in some cases where an ideal semantic merge produces a proper result. The worst acceptable result is: $F(B_M) \sqsubseteq F_M$. In this situation, the result of the change-merge is also compatible with the ideal result, but may diverge in some cases where the semantic merge has a proper value. This situation is less desirable, because it cannot be detected at merge time. This result is the weakest reliable one, which says that the merge produces a correct result whenever it produces a proper result.[5] The approximation argument given in [Ref. 10] for two input merges is also applicable to change-merges. We have discovered that the result of the change-merge is semantically correct in most cases, but we have not proved correctness in all possible cases, so that complete correctness is not guaranteed. We conjecture that the model satisfies the property, $\forall x \in \text{Domain}\ (\perp \neq F(G_M) \neq \top) \Longrightarrow F(G_M)(x) = F_M(x)$, which is halfway between the best pratically realizable situation and the worst acceptable situation.

E. SUMMARY

In this chapter, we have described a model for change-merging PSDL programs. This model divides the problem into three distinct sub-problems and tackles them individually. What we have found is that when broken down, these problems are relatively simple. Simple set operations are used in most cases to perform the change-merge operation. Once the change-merge operation is performed, a consistency check can be performed to ensure that the change-merged implementation accurately represents the change-merged specification. Semantically, this model provides a correct merge in most cases, but the correct result is not guaranteed.

[5] Proper results are normal data values, produced by computations that terminate cleanly.

IV. CONCLUSION

Software is always changing. Whether the change is part of the normal evolution process, or is necessary to fix a problem, it is still a complicated process. The larger software systems become, the more complicated making changes becomes.

A. BENEFITS TO SOFTWARE ENGINEERING

New methods of software development have emerged, such as Rapid Prototyping, which require the ability to quickly create and adapt a prototype to meet the user's needs. This, in turn, makes it necessary to make changes very rapidly. The only way to effectively make changes rapidly, especially to very large systems, is through automation.

The need for automatically making changes arises when a series of software systems have been developed from a common base system, and a change has to be made to the base. If an automated way of propagating the change through all the versions is available, then the software maintainer will save a lot of time, and the end product will probably have fewer errors. This thesis has been directed at defining a method for doing this automatic change propagation.

B. BENEFITS TO CAPS RESEARCH

The Computer Aided Prototyping System, being developed at the Naval Postgraduate School, is a rapid prototyping system. The language used in CAPS, PSDL, is very adaptable to automatic change propagation. We have developed a model for merging three versions of a PSDL program, a base version, and two modifications. We call this three input merge operation, "change-merging". This model can be used, not only for automatically propagating changes through a series of software systems, but can be used to combine the characteristics of two different software systems, which were developed from a common base.

The model is effective for change-merging different versions of a common PSDL program, and we have shown that as long as the result of the change-merge is not an inconsistency, the merged implementation will correctly represent the merged specification. The semantic result of the change-merge has been shown to be correct most of the time.

This model theoretically provides the ability for the engineer writing a prototype, or a series of prototypes, to develop a change to the base version and press a button to invoke the merging mechanism to automatically update all versions of the system. It can also be used to automatically update a series of prototypes in the software base, developed from a common base.

C. FURTHER RESEARCH

In this thesis, we have provided a model on which the change-merger can be based for programs written in PSDL. Our work leads to a method that produces correct results most of the time, but the degree of correctness has not been formally established. Future work should classify the results by how close they come to an ideal semantic merge, suggest stronger approximations that produce results even closer to the ideal semantic merge, and prove the partial correctness of the results.

Eventually, an attempt should be made to develop a change-merger for Ada code, so that data types and PSDL operators implemented in Ada code can be included in the change-merger. For the change-merger to become a reality as currently modelled, a high level language description and specification must be developed, and eventually, the program must be coded in a programming language such as Ada.

APPENDIX A. PSDL GRAMMAR

This grammar uses standard symbology conventions. {Curly Braces} enclose items which may appear zero or more times. [Square Brackets] enclose items which may appear zero or one time. **Bold Face** items are terminal keywords. Items contained in "Double Quotes" are character literals. The "|" vertical bar indicates a list of options from which no more than one item may be selected. This grammar represents the current version of the PSDL grammar as of 20 June 1990.

Start = psdl

psdl = {component}

component = data_type | operator

data_type = **type** id type_spec type_impl

operator = **operator** id operator_spec operator_impl

type_spec = **specification** [generic_param] [type_decl]
    {**operator** id operator_spec} [functionality] **end**

type_impl = **implementation ada** id "{" text "}" **end**
    | **implementation** type_name
    {**operator** id operator_impl} **end**

operator_spec =
    specification {interface} [functionality] **end**

operator_impl = **implementation ada** id "{" text "}"
    | **implementation** psdl_impl

type_decl =
id_list ":" type_name {"," id_list ":" type_name}

functionality = [keywords] [informal_desc] [formal_desc]

psdl_impl = data_flow_diagram [streams] [timers]

[control_constraints] [informal_desc] **end**

type_name = id |

id "[" actual_parameter_list "]" |

id "[" type_decl "]"

actual_parameter_list = actual_parameter

{ "," actual_parameter }

actual_parameter = type_name | expression

interface = attribute [reqmts_trace]

id_list = id {"," id}

keywords = **keywords** id_list

informal_desc = **description** "{" text "}"

formal_desc = **axioms** "{" text "}"

data_flow_diagram = **graph** {vertex} {edge}

streams = **data stream** type_decl

timers = **timer** id_list

attribute = input
| output
| generic_param
| states
| exceptions
| timing_info

input = **input** type_decl

output = **output** type_decl

generic_param = **generic** type_decl

states = **states** type_decl **initially** expression_list

```
exceptions = exceptions id_list

timing_info = [maximum execution time time]
     [minimum calling period time]
     [maximum response time time]

reqmts_trace = by requirements id_list

vertex = vertex op_id [":"time]

edge = edge id [":"time] op_id "->" op_id

op_id = id ["(" [id_list] "|" [id_list] ")"]

control_constraints = control constraints {constraint}

constraint = operator id
     [triggered (trigger | [trigger] if predicate)
[reqmts_trace]]
     [period time [reqmts_trace]]
     [finish within time [reqmts_trace]]
     {constraint_options}

trigger = by all id_list
     | by some id_list

constraint_options =
       output id_list if predicate [reqmts_trace]
     | exception id [if predicate] [reqmts_trace]
     | timer_op id [if predicate] [reqmts_trace]

timer_op = read timer
     | reset timer
     | start timer
     | stop timer

expression_list = expression {"," expression}

time = integer [unit]

unit = ms | sec | min | hours

expression = constant
     | id
     | type_name "." id "(" expression_list ")"

predicate = simple_expression
     | simple_expression rel_op simple_expression
```

```
simple_expression = [sign] integer [unit]
      | [sign] real
      | [not] id
      | string
      | [not] "(" predicate ")"
      | [not] boolean_constant

bool_op = and | or

rel_op = "<" | "<=" | ">" | ">=" | "=" | "/=" | ":"

real = integer "." integer

integer = digit{digit}

boolean_constant = true | false

numeric_constant = real | integer

constant = numeric_constant | boolean_constant

sign = "+" | "-"

char = any printable character except "}"

digit = "0 .. 9"

letter = "a .. z" | "A .. Z" | "_"

alpha_numeric = letter | digit

id = letter{alpha_numeric}

string = """ text """

text = {char}
```

APPENDIX B. PROOF: CONSISTENCY OF CHANGE-MERGE THEOREM

This appendix contains the proof of our Consistency of Change-merge theorem in Chapter III, Figure 35.

PROOF:

Let M = A[Base]B be the result ınt operator after a change-merge operation.

Assume: ***Imp***(G_A, Sp_A), ***Imp***(G_{Base}, Sp_{Base}) and ***Imp***(G_B, Sp_B).

Need to Show: ***Imp***(G_M, Sp_M).

$(\Longrightarrow)$

Assume $x \in I_M$, $y \in O_M$, and $z \in St_M$.

Need to Show A. $(x \in M.S \wedge (EXT, x) \in M.LI)$

B. $(y \in M.S \wedge (y, EXT) \in M.LO)$

C. $(z \in DS_M \wedge \exists w \in M.V\ [(w, z) \in M.LI]$

$\wedge \exists v \in M.V\ [(z, v) \in M.LO]$

A. There are two possibilities: $x \in I_{Base}$ and $\neg(x \in I_{Base})$.

Case 1: $x \in I_{Base}$

Then by definition of change-merge, $x \in I_A \wedge x \in I_B$.

Since ***Imp***(G_{Base}, Sp_{Base}), ***Imp***(G_A, Sp_A) and ***Imp***(G_B, Sp_B),

Then $x \in Base.S \wedge (EXT, x) \in Base.LI) \wedge$

$x \in A.S \wedge (EXT, x) \in A.LI) \wedge$

$x \in B.S \wedge (EXT, x) \in B.LI)$

And by definition of change-merge

$x \in M.S \wedge (EXT, x) \in M.LI)$ (**A.**)

Case 2: $\neg(x \in I_{Base})$

Then by definition of change-merge, $x \in I_A \vee x \in I_B$.

Since ***Imp***(G_{Base}, Sp_{Base}), ***Imp***(G_A, Sp_A) and ***Imp***(G_B, Sp_B),

Then $\neg(x \in \text{Base.S} \wedge (\text{EXT}, x) \in \text{Base.LI})) \wedge$
$((x \in \text{A.S} \wedge (\text{EXT}, x) \in \text{A.LI}) \vee$
$(x \in \text{B.S} \wedge (\text{EXT}, x) \in \text{B.LI}))$

Since $(\text{EXT}, x) \in \text{Base.LI} \Rightarrow x \in \text{Base.S}$

Then $x \in \text{Base.S} \wedge (\text{EXT}, x) \in \text{Base.LI} \Leftrightarrow$
$(\text{EXT}, x) \in \text{Base.LI}$

Thus $\neg(x \in \text{Base.S} \wedge (\text{EXT}, x) \in \text{Base.LI})) \Leftrightarrow$
$\neg(\text{EXT}, x) \in \text{Base.LI}$

Thus $(\text{EXT}, x) \in \text{A.LI} \vee (\text{EXT}, x) \in \text{B.LI}$

And $x \in \text{A.S} \vee x \in \text{B.S}$

And by definition of change-merge
$x \in \text{M.S} \wedge (\text{EXT}, x) \in \text{M.LI})$ (**A.**)

B. There are two possibilities: $y \in O_{Base}$ and $\neg(y \in O_{Base})$.

Case 1: $y \in O_{Base}$

Then by definition of change-merge, $y \in O_A \wedge y \in O_B$.

Since ***Imp***(G_{Base}, Sp_{Base}), ***Imp***(G_A, Sp_A) and ***Imp***(G_B, Sp_B),

Then $y \in \text{Base.S} \wedge (y, \text{EXT}) \in \text{Base.LO}) \wedge$
$y \in \text{A.S} \wedge (y, \text{EXT}) \in \text{A.LO}) \wedge$
$y \in \text{B.S} \wedge (y, \text{EXT}) \in \text{B.LO})$

And by definition of change-merge
$y \in \text{M.S} \wedge (y, \text{EXT}) \in \text{M.LO})$ (**B.**)

Case 2: $\neg(y \in O_{\text{Base}})$

Then by definition of change-merge, $y \in O_{A} \vee y \in O_{B}$.

Since ***Imp***$(G_{\text{Base}}, Sp_{\text{Base}})$, ***Imp***$(G_{A}, Sp_{A})$ and ***Imp***(G_{B}, Sp_{B}),

Then $\neg(y \in \text{Base.S} \wedge (y,\text{EXT}) \in \text{Base.LO})) \wedge$
$((y \in \text{A.S} \wedge (y,\text{EXT}) \in \text{A.LO}) \vee$
$(y \in \text{B.S} \wedge (y,\text{EXT}) \in \text{B.LO}))$

Since $(y,\text{EXT}) \in \text{Base.LO} \Longrightarrow y \in \text{Base.S}$

Then $y \in \text{Base.S} \wedge (y,\text{EXT}) \in \text{Base.LO} \Longleftrightarrow$
$(y,\text{EXT}) \in \text{Base.LO}$

Thus $\neg(y \in \text{Base.S} \wedge (y,\text{EXT}) \in \text{Base.LO})) \Longleftrightarrow$
$\neg(y,\text{EXT}) \in \text{Base.LO}$

Thus $(y,\text{EXT}) \in \text{A.LO} \vee (y,\text{EXT}) \in \text{B.LO}$

And $y \in \text{A.S} \vee y \in \text{B.S}$

And by definition of change-merge
$y \in \text{M.S} \wedge (y,\text{EXT}) \in \text{M.LO})$ (**B**.)

C. There are two possibilities: $z \in St_{\text{Base}}$ and $\neg(z \in St_{\text{Base}})$.

Case 1: $z \in St_{\text{Base}}$

Then by definition of change-merge, $z \in St_{A} \wedge z \in St_{B}$.

Since ***Imp***$(G_{\text{Base}}, Sp_{\text{Base}})$, ***Imp***$(G_{A}, Sp_{A})$ and ***Imp***(G_{B}, Sp_{B}),

Then $(z \in DS_{\text{Base}} \wedge \exists w \in \text{Base.V}\ [(w, z) \in \text{Base.LI}]$
$\wedge \exists v \in \text{Base.V}\ [(z, v) \in \text{Base.LO}]) \wedge$
$(z \in DS_{A} \wedge \exists w \in \text{A.V}\ [(w, z) \in \text{A.LI}]$
$\wedge \exists v \in \text{A.V}\ [(z, v) \in \text{A.LO}]) \wedge$
$(z \in DS_{B} \wedge \exists w \in \text{B.V}\ [(w, z) \in \text{B.LI}]$
$\wedge \exists v \in \text{B.V}\ [(z, v) \in \text{B.LO}])$

And by definition of change-merge

$$\begin{aligned}&(z \in DS_M \wedge \exists w \in M.V\ [(w, z) \in M.LI]\\&\qquad \wedge \exists v \in M.V\ [(z, v) \in M.LO])\ \text{(C.)}\end{aligned}$$

Case 2: $\neg(z \in St_{Base})$

Then by definition of change-merge, $z \in St_A \vee z \in St_B$.

Since $\textit{Imp}(G_{Base}, Sp_{Base})$, $\textit{Imp}(G_A, Sp_A)$ and $\textit{Imp}(G_B, Sp_B)$,

$$\begin{aligned}\text{Then } &\neg(z \in DS_{Base} \wedge \exists w \in Base.V\ [(w, z) \in Base.LI]\\&\qquad \wedge \exists v \in Base.V\ [(z, v) \in Base.LO]) \wedge\\&((z \in DS_A \wedge \exists w \in A.V\ [(w, z) \in A.LI]\\&\qquad \wedge \exists v \in A.V\ [(z, v) \in A.LO]) \vee\\&(z \in DS_B \wedge \exists w \in B.V\ [(w, z) \in B.LI]\\&\qquad \wedge \exists v \in B.V\ [(z, v) \in B.LO]))\end{aligned}$$

And by definition of change-merge

$$\begin{aligned}&(z \in DS_M \wedge \exists w \in M.V\ [(w, z) \in M.LI]\\&\qquad \wedge \exists v \in M.V\ [(z, v) \in M.LO])\ \text{(C.)}\end{aligned}$$

Therefore by A, B, and C we conclude

$$\begin{aligned}&x \in I_M \Longrightarrow (x \in M.S \wedge (EXT, x) \in M.LI) \wedge\\&y \in O_M \Longrightarrow (y \in M.S \wedge (y, EXT) \in M.LO) \wedge\\&z \in St_M \Longrightarrow (z \in DS_M \wedge \exists w \in M.V\ [(w, z) \in M.LI]\\&\qquad \wedge \exists v \in M.V\ [(z, v) \in M.LO]\end{aligned}$$

($\Longleftarrow$)

Assume $(x \in M.S \wedge (EXT, x) \in M.LI)$ and

$(y \in M.S \wedge (y, EXT) \in M.LO)$

Need to Show D. $x \in I_M$

E. $y \in O_M$

D. There are two possibilities:

$x \in$ Base.S $\wedge$ (EXT, x) $\in$ Base.LI and

$\neg(x \in$ Base.S $\wedge$ (EXT, x) $\in$ Base.LI).

Case 1: $x \in$ Base.S $\wedge$ (EXT, x) $\in$ Base.LI

Then by definition of change-merge,

$x \in$ A.S $\wedge$ (EXT, x) $\in$ A.LI $\wedge$

$x \in$ B.S $\wedge$ (EXT, x) $\in$ B.LI

Since ***Imp***(G_{Base}, Sp_{Base}), ***Imp***(G_A, Sp_A) and ***Imp***(G_B, Sp_B),

$x \in I_{Base} \wedge x \in I_A \wedge x \in I_B$

And by definition of change-merge, $x \in I_M$. (**D**.)

Case 2: $\neg(x \in$ Base.S $\wedge$ (EXT, x) $\in$ Base.LI)

Since (EXT, x) $\in$ Base.LI $\Longrightarrow x \in$ Base.S

Then $x \in$ Base.S $\wedge$ (EXT, x) $\in$ Base.LI $\Longleftrightarrow$
(EXT, x) $\in$ Base.LI

Thus $\neg(x \in$ Base.S $\wedge$ (EXT, x) $\in$ Base.LI)) $\Longleftrightarrow$
$\neg$(EXT, x) $\in$ Base.LI

Thus (EXT, x) $\in$ A.LI $\vee$ (EXT, x) $\in$ B.LI

And $x \in$ A.S $\vee$ $x \in$ B.S

Then $x \in$ A.S $\wedge$ (EXT, x) $\in$ A.LI $\vee$
$x \in$ B.S $\wedge$ (EXT, x) $\in$ B.LI

Since ***Imp***(G_{Base}, Sp_{Base}), ***Imp***(G_A, Sp_A) and ***Imp***(G_B, Sp_B),

$\neg(x \in I_{Base}) \wedge (x \in I_A \vee x \in I_B)$

And by definition of change-merge, $x \in I_M$. (**D**.)

E. There are two possibilities:

$y \in \text{Base.S} \wedge (y,\text{EXT}) \in \text{Base.LO}$ and

$\neg(y \in \text{Base.S} \wedge (y,\text{EXT}) \in \text{Base.LO})$.

Case 1: $y \in \text{Base.S} \wedge (y,\text{EXT}) \in \text{Base.LO}$

Then by definition of change-merge,

$y \in \text{A.S} \wedge (y,\text{EXT}) \in \text{A.LO} \wedge$

$y \in \text{B.S} \wedge (y,\text{EXT}) \in \text{B.LO}$

Since $\boldsymbol{Imp}(G_{Base}, Sp_{Base})$, $\boldsymbol{Imp}(G_A, Sp_A)$ and $\boldsymbol{Imp}(G_B, Sp_B)$,

$y \in I_{Base} \wedge y \in I_A \wedge y \in I_B$

And by definition of change-merge, $y \in I_M$. (**E.**)

Case 2: $\neg(y \in \text{Base.S} \wedge (y,\text{EXT}) \in \text{Base.LO})$

Since $(y,\text{EXT}) \in \text{Base.LO} \Rightarrow y \in \text{Base.S}$

Then $y \in \text{Base.S} \wedge (y,\text{EXT}) \in \text{Base.LO} \Leftrightarrow$
$(y,\text{EXT}) \in \text{Base.LO}$

Thus $\neg(y \in \text{Base.S} \wedge (y,\text{EXT}) \in \text{Base.LO})) \Leftrightarrow$
$\neg(y,\text{EXT}) \in \text{Base.LO}$

Thus $(y,\text{EXT}) \in \text{A.LO} \vee (y,\text{EXT}) \in \text{B.LO}$

And $y \in \text{A.S} \vee y \in \text{B.S}$

Then $y \in \text{A.S} \wedge (y,\text{EXT}) \in \text{A.LO} \vee$
$y \in \text{B.S} \wedge (y,\text{EXT}) \in \text{B.LO}$

Since $\boldsymbol{Imp}(G_{Base}, Sp_{Base})$, $\boldsymbol{Imp}(G_A, Sp_A)$ and $\boldsymbol{Imp}(G_B, Sp_B)$,

$\neg(y \in I_{Base}) \wedge (y \in I_B \vee y \in I_B)$

And by definition of change-merge, $y \in I_M$. (**E.**)

Therefore by D and E we conclude

$$(x \in \mathrm{M.S} \wedge (\mathrm{EXT},\ x) \in \mathrm{M.LI}) \Rightarrow x \in \mathrm{I_M} \wedge$$
$$(y \in \mathrm{M.S} \wedge (y,\ \mathrm{EXT}) \in \mathrm{M.LO}) \Rightarrow y \in \mathrm{O_M}$$

Therefore By ($\Rightarrow$) and ($\Leftarrow$), ***Imp***(G_M, Sp_M). □

LIST OF REFERENCES

1. Berzins, V., *Unpublished Class Notes from CS4500*, January 1990, Naval Postgraduate School, Monterey, California 93940.

2. Fountain, H., *Rapid Prototyping: A Survey of Methodologies and Models*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1990.

3. Schneidewind, N., "The State of Software Maintenance", *IEEE Transactions on Software Engineering*, pp. 303-310, March 1987.

4. Borison, E., "A Model of Software Manufacture", Proceedings of an International Workshop, Trondheim, Norway, pp. 197-220, Springer-Verlag, June 1986.

5. Naval Postgraduate School, Report NPS52-90-014 , *A Graph Model of Software Maintenance*, by Mostov, I., Luqi, and Hefner, K., 1989.

6. Luqi, "Software Evolution Through Rapid Prototyping", *IEEE Computer*, May 1989.

7. Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real Time Software", *IEEE Transactions on Software Engineering*, pp.1409-1423, October 1988.

8. White, L., *The Development of a Rapid Prototyping Environment*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1989.

9. Altizer, C., *Implementation of a Language Translator for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

10. Berzins, V., "On Merging Software Extensions", *Acta Informatica*, Vol. 23, pp. 607-619, 14 July 1986.

11. Horwitz, S., Prins, J., and Reps, T., "Integrating Non-Interfering Versions of Programs", *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, New York, New York, 13 - 15 January 1988.

GLOSSARY

Approximates - A program P approximates another program Q if P is defined everywhere that Q is defined. Q may or may not be defined in places where P is undefined.

Bipartite - A bipartite graph is a graph in which the vertices can be divided into distinct sets, where there exists no edge from one vertex to another within the same set.

Computer-Aided Prototyping System(CAPS) - This system is being developed by Dr. Berzins and Dr. Luqi at the Naval Postgraduate School for use in Rapid Prototyping of Real-Time Systems.

CPU - Central Processor Unit

Directed Acyclic Graph(DAG) - It is a graph which contains directed edges and no cycles.

Feasible - Any PDG G_p is feasible if it is a PDG for a program P. The slice G/S is feasible if $S \subseteq V(G)$ and G is feasible.

FIFO - First In First Out.

Greatest Common Approximation - The greatest common approximation of two programs is considered to be a program which approximates both programs and contains all of the functionality common to both programs.

Least Common Extension - The least common extension of two programs is the result of merging the functionality contained in both programs. Both input programs approximate the least common extension.

Program Dependence Graph(PDG) - Used by Horwitz et al. to abstractly represent programs. It consists of a set of vertices and a set of edges which link these vertices. The vertices represent operations performed in the program, and the edges represent control flow and data flow dependences between those operations.

Prototyping System Description Language(PSDL) - Rapid prototyping language developed by Dr. Luqi at the Naval Postgraduate School for use in designing prototypes within the CAPS system.

Software Maintenance - All activities performed on a software system during its life time to enhance or repair the system.

Software Manufacture - The combining of primitive components of a software system, through a sequence of derivations, into one or more software products.

BIBLIOGRAPHY

Naval Postgraduate School, Report NPS52-89-023, *Computer Languages for Rapid Prototyping*, by Luqi, March 1989.

Naval Postgraduate School, Report NPS52-90-004, *Maintenance Problems in Military Software Systems*, by Mostov, I. and Luqi, August 1989.

University of Oregon, CIS-TR-89-02, *Automating the Parallel Elaboration of Specifications: Preliminary Findings*, by W. N. Robinson, 7 February 1989.

University of Oregon, CIS-TR-89-03, *Integrating Multiple Specifications Using Domain Goals*, by W. N. Robinson, 23 February 1989.

University of Oregon, CIS-TR-89-13, *Negotiation Behavior During Multiple Agent Specification: A Need for Automated Conflict Resolution*, by W. N. Robinson, 6 September 1989.

University of Wisconsin-Madison, Computer Sciences Technical Report 856, *On the Algebraic Properties of Program Integration*, by T. Reps, June 1989.

Tanik, M. and Yeh, R., "Rapid Prototyping in Software Development", *Computer*, vol. 22, pp. 9-10, May 1989.

Wilde, N., Huitt, R. and Huitt, S., "Dependency Analysis Tools: Reusable Components for Software Maintenance", *Proceedings of the Conference on Software Maintenance - 1989*, IEEE, Washington, District of Columbia, 16 - 19 October 1989.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, Virginia 22304-6145

Superintendent 2
Attn: Library, Code 0142
Naval Postgraduate School
Monterey, California 93943-5000

Superintendent 1
Attn: Director of Research Administration, Code 012
Naval Postgraduate School
Monterey, California 93943-5000

Chairman, Code 52 1
Computer Science Department
Naval Postgraduate School
Monterey, California 93943

Office of Naval Research 1
800 N. Quincy Street
Arlington, Virginia 22217-5000

Center for Naval Analysis 1
4401 Ford Avenue
Alexandria, Virginia 22302-0268

National Science Foundation 1
Division of Computer and Computation Research
Washington, District of Columbia 20550

OP-941 1
Office of the Chief of Naval Operations
Washington, District of Columbia 20350

OP-945 1
Office of the Chief of Naval Operations
Washington, District of Columbia 20350

Commander 2
Naval Telecommunications Command
4401 Massachusetts Avenue NW
Washington, District of Columbia 20390-5290

Commander 1
Naval Data Automation Command
Washington Navy Yard
Washington, District of Columbia 20374-1662

Dr. Lui Sha 1
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15260

Colonel C. Cox, USAF 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, District of Columbia 20318-8000

Commander 1
Code 5150
Naval Research Laboratory
Washington, District of Columbia 20375-5000

Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

Defense Advanced Research Projects Agency (DARPA) 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

Defense Advanced Research Projects Agency (DARPA) 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

Dr. R. M. Carroll (OP-01B2) 1
Office of the Chief of Naval Operations
Washington, District of Columbia 20350

Dr. Aimram Yehudai 1
School of Mathematical Sciences
Tel Aviv University
Tel Aviv, Israel 69978

Dr. Robert M. Balzer 1
Information Sciences Institute
University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Ray, California 90292-6695

Dr. Ted Lewis 1
Editor-in-Chief, IEEE Software
Computer Science Department
Oregon State University
Corvallis, Oregon 97331

Dr. R. T. Yeh 1
International Software Systems Inc.
12710 Research Boulevard, Suite 301
Austin, Texas 78759

Dr. C. Green 1
Kestrel Institute
1801 Page Mill Road
Palo Alto, California 94304

Professor D. Berry 1
Department of Computer Science
University of California
Los Angeles, California 90024

Director 1
Naval Telecommunications System Integration Center
NAVCOMMUNIT Washington
Washington, District of Columbia 20363-5110

Dr. Knudsen 1
Code PD50
Space and Naval Warfare Systems Command
Washington, District of Columbia 20363-5110

Ada Joint Program Office 1
ATTN: OUSDRE(R&AT)
Pentagon
Washington, District of Columbia 23030

Captain A. Thompson, United States Navy 1
Naval Sea Systems Command
National Center #2, Suite 7N06
Washington, District of Columbia 22202

Dr. Peter Ng 1
Computer Science Department
New Jersey Institute of Technology
Newark, New Jersey 07102

Dr. Van Tilborg 1
Office of Naval Research
Computer Science Division, Code 1133
800 North Quincy Street
Arlington, Virginia 22217-5000

Dr. R. Wachter 1
Office of Naval Research
Computer Science Division, Code 1133
800 North Quincy Street
Arlington, Virginia 22217-5000

Dr. J. Smith, Code 121 1
Office of Naval Research
Applied Mathematics and Computer Science
800 North Quincy Street
Arlington, Virginia 22217-5000

Dr. R. Kieburtz 1
Oregon Graduate Center Portland (Beaverton)
Portland, Oregon 97005

Dr. M. Ketabchi 1
Department of Electrical Engineering and Computer Science
Santa Clara University
Santa Clara, California 95053

Dr. L. Belady 1
Software Group, MCC
9430 Research Boulevard
Austin, Texas 78759

Dr. Murat Tanik 1
Computer Science and Engineering Department
Southern Methodist University
Dallas, Texas 75275

Dr. Ming Liu 1
Department of Computer and Information Science
Ohio State University
2036 Neil Avenue Mall
Columbus, Ohio 43210-1277

Mr. William E. Rzepka 1
U.S. Air Force Systems Command
Rome Air Development Center
ATTN: RADC/COE
Griffis Air Force Base, New York 13441-5700

Dr. C.V. Ramamoorthy 1
Department of Electrical Engineering and Computer Science
Computer Science Division
University of California at Berkeley
Berkeley, California 90024

Dr. Nancy Levenson 1
Department of Computer and Information Science
University of California at Irvine
Irvine, California 92717

Dr. Mike Reiley 1
Fleet Combat Directional Systems Support Activity
San Diego, California 92147-5081

Dr. William Howden 1
Department of Computer Science
University of California at San Diego
La Jolla, California 92093

Dr. Earl Chavis (OP-162) 1
Office of the Chief of Naval Operations
Washington, District of Columbia 20350

Dr. Jane W. S. Liu 1
Department of Computer Science
University of Illinois
Urbana Champaign, Illinois 61801

Dr. Alan Hevner 1
College of Business Management
Tydings Hall, Room 0137
University of Maryland
College Park, Maryland 20742

Dr. Y. H. Chu 1
Computer Science Department
University of Maryland
College Park, Maryland 20742

Dr. N. Roussapoulos 1
Computer Science Department
University of Maryland
College Park, Maryland 20742

Dr. Alfs Berztiss 1
Department of Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania 15260

Dr. Al Mok 1
Computer Science Department
University of Texas at Austin
Austin, Texas 78712

George Sumiall 1
Headquarters
U.S. Army Communications and Electronics Command
ATTN: AMSEL-RD-SE-AST-SE
Fort Monmouth, New Jersey 07703-5000

Mr. Joel Trimble 1
1211 South Fern Street, C107
Arlington, Virginia 22202

Dr. Linwood Sutton 1
Code 423
Naval Ocean Systems Center
San Diego, California 92152-5000

Dr. Sherman Gee 1
Code 221
Office of Naval Technology
200 North Quincy Street
Arlington, Virginia 22217

Dr. Mario Barbacci 1
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Dr. Mark Kellner 1
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Dr. Valdis Berzins 10
Code 52Be
Computer Science Department
Naval Postgraduate School
Monterey, California 93943

Dr. Luqi 2
Code 52Lq
Computer Science Department
Naval Postgraduate School
Monterey, California 93943

Captain David A. Dampier, United States Army 2
8508 Hopewell
El Paso, Texas 79925